2 Erweiterte Syntax

Die Welt wurde aus der Leere gebildet,
wie Utensilien aus einem Holzblock.
Der Herr kennt die Utensilien,
hält sich jedoch an den Block:
damit kann er alle Dinge benutzen.

Wie in Kapitel 1 erwähnt, glauben wir, dass wir zur wirklichen Beherrschung einer Programmiersprache nicht nur deren Syntax und Semantik verstehen müssen, sondern auch ihre Philosophie, den Hintergrund und die Hauptmerkmale der Struktur. Um PHP zu meistern, müssen Sie auch alle seine Eigenheiten gut kennen.

2.1 PHP-Syntax

PHP ist eine Mischung aus verschiedenen Sprachen. Sie können einen starken Einfluss von C erkennen (einige sagen Java, aber Java ist auch aus C entstanden). Während die Syntax von PHP stark durch C beeinflusst ist, unterscheidet sich die PHP-Semantik von der von C sehr stark. PHP wird interpretiert und hat keine strengen Variablentypen. Wenn Sie auf eine Variable referenzieren, wird sein Typ im »Vorübergehen« ermittelt und so behandelt, wie es die gegenwärtige Situation erfordert. Dies ist zwar eine etwas vereinfachte Erklärung, Sie sollten sie aber während der Entwicklung im Hinterkopf behalten.

PHP ist eine interpretierte Sprache, in welcher der Programmcode ausgewertet und Schritt für Schritt ausgeführt wird. Zusammen mit der Art und Weise, wie PHP Variablen verwaltet, erhalten Sie damit vielfältige Möglichkeiten für die Programmierung, jedoch auch eine Menge Fallen. Dieses Kapitel befasst sich mit den typischen Dingen, die man tun bzw. lassen sollte, um erweiterte syntaktische und Algorithmenfunktionen zu erhalten:

- Definition von Konstanten
- Array-Funktionen
- Klassen
- Verknüpfte Listen
- Bäume
- Assoziative Arrays
- Mehrdimensionale Arrays
- Variable Argumente und variable Argumentenlisten

- Variable Variablennamen
- Variable Funktionsnamen
- Selbstmodifizierender Highlevel-Code
- Polymorphisierung

Auf all diese Themen werden wir in den folgenden Abschnitten eingehen.

2.2 Definition von Konstanten

In PHP gibt es zwar keinen Ausdruck und keine Konstruktion, um Konstanten in Form von nicht-modifizierbaren Variablen zu definieren, trotzdem können Sie dieses Ziel durch Verwendung von definierten Werten erreichen. Definierte Werte sollten verwendet werden, um alle festen Werte zu ersetzen, wie z. B. Fehlercodes, Dateiformatkonstanten, spezielle Zeichenfolgen und andere Konstrukte, die eine spezielle Bedeutung für das Programm oder die Bibliothek haben und während der Programmausführung nicht geändert werden sollen.

Definierte Werte haben den großen Vorteil, dass sie die Bedeutung spezieller Werte verdeutlichen und gleichzeitig eine weitere Abstraktionsebene bieten:

```
// read file type from input
$file_type = fgets($file, 32);

// decide what kind of file it is
switch($file_type)
{
    case FT_GIF_IMAGE:   /* handle GIFs here */
                         break;

    case FT_PNG_IMAGE:   /* handle PNGs here */
                         break;

    case FT_ZIP_ARCHIVE: /* handle ZIPs here */
                         break;
}
```

Hinweis: Mit dem Begriff spezielle Werte meinen wir »magische Nummern« oder spezielle Zeichenfolgen. Wenn Sie beispielsweise aus Ihrem Programm auf eine GIF-Datei zugreifen wollen und Ihr Programm intern GIF durch die »magische Nummer« 6 erkennt (weil dies der Programmierer so festgelegt hatte – es könnte auch 1234 sein), können Sie mit `define("GIF_FILE", 6)` für diesen Wert eine Definition erzeugen und künftig auf Ihre »magische Nummer« mit dem Schlüsselwort `GIF_FILE` zugreifen.

Dieser Programmabschnitt liest einen Bezeichner aus einer Eingabedatei und entscheidet dann, wie er auf diesen Bezeichner reagiert. Der Bezeichner gibt an, ob es sich bei den folgenden Daten um ein GIF-Bild, ein PNG-Bild oder ein ZIP-Archiv handelt. Er könnte folgendermaßen aussehen:

GIF-Bild »GIF_IMG«

PNG-Bild »PNG_IMG«

ZIP-Archiv »ZIP_ARC«

Diese Bezeichner würden in einer include-Datei folgendermaßen definiert werden:

```
define("FT_GIF_IMAGE", "GIF_IMG");
define("FT_PNG_IMAGE", "PNG_IMG");
define("FT_ZIP_ARCHIVE", "ZIP_ARC");
```

Diese Einrichtung hat den Vorteil, dass Sie alle Bezeichner an einer zentralen Stelle sammeln können. Wenn Sie einen Bezeichner ändern müssen, brauchen Sie nur seine Definition ändern. Andernfalls müssten Sie den gesamten Programmcode durchsuchen und jede dieser Zeichenfolgen manuell ersetzen. Mithilfe definierter Werte lässt sich diese Arbeit auf eine einzelne Programmzeile reduzieren.

Die Namen der definierten Werte sollten stets in Großbuchstaben geschrieben werden, um sie als solche zu kennzeichnen. In einigen Fällen, wie bei Bibliotheksfunktionen, sollte ihnen außerdem ein Präfix vorangestellt sein. Im vorangegangenen Beispiel haben die Bezeichner das Präfix »FT_« für »file-type« (Dateityp).

Versuchen Sie, definierte Werte so häufig wie möglich einzusetzen. Wenn Sie auf eine Situation treffen, in der Sie normalerweise einen Wert in Ihr Programm fest eingeben würden, sollten Sie bedenken, dass die Eingabe von fixen Werten unvorteilhaft ist. Betriebssysteme oder ähnliche maschinencodeorientierte Programme haben häufig die längsten Definitionslisten, denn um sie portabel zu gestalten, muss jede Kleinigkeit abstrahiert werden. Sie können keine Annahmen über die Byte-Größe, die Wortlänge oder die Registergröße machen; im Prinzip muss alles, was Sie sehen, auf irgendeine Weise abstrahiert werden. Da PHP per se portierbar ist, bedeutet dies, dass es an keine bestimmte Hardware oder andere Umgebungskonfigurationen gebunden ist (sein Interpreter ändert die Umgebung nicht, wie auch immer das zugrunde liegende Betriebssystem aussehen mag). Daher brauchen Sie mit Definitionen in PHP nicht übertreiben. Aber ein gutes Maß zeugt von guten Programmierstil.

2.3 Array-Funktionen

Die wichtigsten Array-Funktionen sind `list()`, `each()` und `count()`.

`list()` ist eine Art Operator, der aus einer Reihe von Variablen einen l-Wert bildet (einen Wert, der auf der linken Seite eines Ausdrucks verwendet werden kann), der sich ähnlich wie ein Element eines mehrdimensionalen Arrays wiederum als neue Entität darstellt. Als Argument kann er eine Liste von Variablen besitzen. Wird ihm etwas zugeordnet (eine Liste von Variablen oder ein Array-Element), dann wird die Liste der Variablen, die als Argument des Operators `list()` angegeben wurde, von links nach rechts abgearbeitet. Diesen Argumenten wird dann ein entsprechender Wert aus dem r-Wert (einem Wert, der auf der rechten Seite eines Ausdrucks verwendet werden kann) zugewiesen. Dies lässt sich am besten anhand eines Beispiels erklären:

```
$result = mysql_db_query($mysql_handle, $mysql_db,
                         "SELECT car_type, car_color, car_speed
                         FROM cars WHERE car_id=$car_id");

list($car_type, $car_color, $car_speed) = mysql_fetch_row($result);
```

Hinweis: Dieser Programmcode wird hier nur als Beispiel verwendet. Es empfiehlt sich nicht, diesen Programmcode in realistische Programme zu integrieren, da er darauf aufbaut, dass die Felder in derselben Reihenfolge bleiben. Wenn Sie die Reihenfolge der Felder ändern, müssen Sie auch die Reihenfolge der Variablen in der Anweisung `list()` ändern. Die Verwendung von assoziativen Arrays und die manuelle Extraktion von Werten bedeutet einen zusätzlichen Programmierungsaufwand, führt aber zu einem stabileren Programmcode. Programmcode, wie der oben gezeigte, sollte nur für Optimierungszwecke implementiert werden.

Die SQL-Abfrage wählt die Werte `car_type`, `car_color` und `car_speed` aus einer Tabelle mit Fahrzeugdaten aus. Das Ergebnis der Abfrage wird anschließend mithilfe von `mysql_fetch_row()` ausgelesen, welches die drei Werte in einem Array zurückgibt. `car_type` wird dabei an die Indexposition 0 geschrieben, `car_color` an Indexposition 1 und `car_speed` an Indexposition 2. Diese Werte werden von links nach rechts gelesen und anschließend Wert für Wert den in der `list()`-Anweisung angegebenen Argumenten zugeordnet.

Damit erhalten Sie die folgenden Zuordnungen:

Listenargument	SQL-Feld
`$car_type`	`car_type` (Array-Position 0)
`$car_color`	`car_color` (Array-Position 1)
`$car_speed`	`car_speed` (Array-Position 2)

Die Anweisung `list()` ist besonders nützlich, wenn Sie eine Sammlung von Werten in einzelne Variablen unterteilen möchten. Dies passiert recht häufig bei der Datenbankprogrammierung. Beachten Sie jedoch, dass `list()` nur als l-Wert und nicht als r-Wert fungieren kann. Sie können `list()` nicht verwenden, um eine Reihe von Variablen auszutauschen. Die folgende Anweisung würde beispielsweise nicht funktionieren:

```
list($var1, $var2) = list($var2, $var1);
```

Die Anweisung `each()` wird häufig mit der Anweisung `list()` verwendet. `each()` arbeitet ein Array ab und gibt die einzelnen Elemente in einer Schlüssel/Wert-Kombination zurück. Dies geschieht durch »Abwandern« des Eingabe-Arrays. PHP weist jedem Array einen internen Zeiger zu. Dieser Zeiger weist zu Beginn auf das erste Element des Arrays. Bei jedem Aufruf von `each()` wird das Element zurückgegeben, auf welches der interne Zeiger weist. Anschließend wird der Zeiger um eine vorgegebene Anzahl von Elementen hochgesetzt.

Das Rückgabeformat des Schlüssel/Wert-Paares ist ein vier Elemente umfassendes Array mit den Schlüsseln »0«, »1«, »key« und »value«. Dies bedeutet, dass es sowohl als indiziertes Array als auch als assoziatives Array verwendet werden kann. Der indexierte Teil des Arrays (mit den Schlüsseln »0« und »1«) enthält den Schlüssel des Quellelements an Indexposition 0; der Wert befindet sich an Indexposition 1. Dieselbe Information erhalten Sie auch über den assoziativen Teil des Arrays. (Genaugenommen ist eine Trennung von assoziativen und indizierten Teil des Arrays hier nicht korrekt, da indizierte Arrays nur eine Sonderform der assoziativen Arrays sind. Theoretisch sind es zwar ganz verschiedene Dinge, aber in der Praxis sind sie in PHP identisch. Weitere Einzelheiten finden Sie weiter unten.) Der Schlüssel des Quellelements ist in »key« angegeben, während der Wert in »value«. steht. Ein Beispiel:

```
$my_array = array("Element 1", "Element 2", "Element 3");
```

Hierdurch wird ein Array mit dem folgenden Inhalt erzeugt:

```
Element 1
Element 2
Element 3
```

Um das Prinzip von `each()` besser zu verstehen, ist es sinnvoll, eine detaillierte Auflistung dieses Arrays zu erstellen:

Schlüssel	Wert
0	Element 1
1	Element 2
2	Element 3

Hier sehen die Auflistung aller Schlüssel/Wert-Paare, die in dem Array `$my_array` enthalten sind:

```
list($key, $value) = each($my_array);
```

Jetzt wenden wir hierauf `each()` an. Beim ersten Aufruf wird das erste vier Elemente umfassende Array zurückgegeben, welches das erste Schlüssel/Wert-Paar aus `$my_array`. enthält. Da wir nur zwei Argumente dem Operators `list()` angegeben haben, können auch nur die Werte aus dem vier-Elemente-Array zugeordnet werden. Es handelt sich um »0«, den ersten Schlüssel, und »Element 1«, das erste Wertelement aus `$my_array`.

Der folgenden Programmcode listet den Inhalt eines Arrays auf:

```
$my_array = array("Element 1", "Element 2", "Element 3");

while(list($key, $value) = each($my_array))
    print("Key: $key, Value: $value<br>");
```

Das Skript erzeugt eine Ausgabe, wie jene, die in Abbildung 2.1. zu sehen ist.

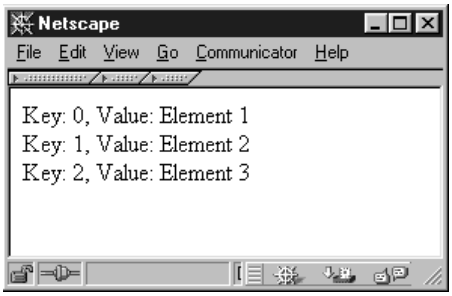

Abbildung 2.1: Array-Auflistung mit `each()`

Sie können `each()` auch verwenden, um die Elemente anzuzeigen, die `each()` selbst zurückgibt:

```
$my_array = array("Element 1", "Element 2", "Element 3");

while($four_element_array = each($my_array))
{
    while(list($key, $value) = each($four_element_array))
        print("Key $key, Value $value<br>");
}
```

Damit erhalten Sie das in Abbildung 2.2 dargestellte Ergebnis.

Sie können deutlich sehen, dass die Einträge »1«, »value«, »0« und »key« in der Ausgabe, sowie »0« und »1« zusammen nur als »key« und »value« verwendet werden sollten. Beachten Sie, dass jedes Paar einen Eintrag des Quell-Arrays darstellt.

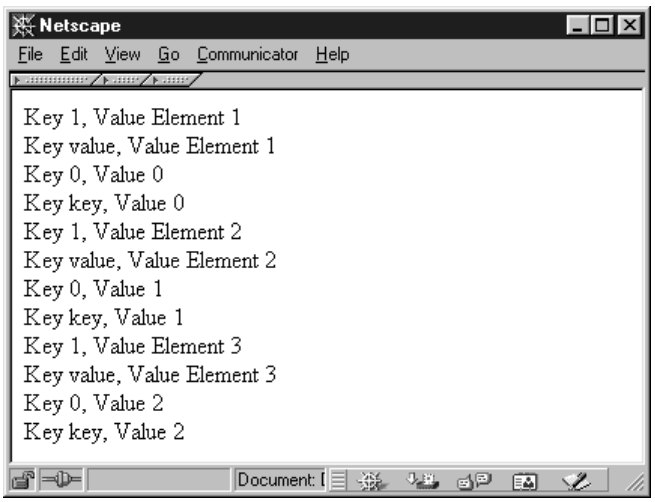
Netscape
File Edit View Go Communicator Help
Key 1, Value Element 1
Key value, Value Element 1
Key 0, Value 0
Key key, Value 0
Key 1, Value Element 2
Key value, Value Element 2
Key 0, Value 1
Key key, Value 1
Key 1, Value Element 3
Key value, Value Element 3
Key 0, Value 2
Key key, Value 2
Document:

Abbildung 2.2: Rückgabewerte von `each()`

`each()` für ein indexiertes Array zu verwenden scheint zunächst keinen Sinn zu machen, da die Elemente eines indexierten Arrays sehr viel leichter mithilfe einer `for()`-Anweisung gelesen werden können. Dabei gibt es jedoch einige Tücken. Zunächst ist in PHP das indexierte Array nur eine Sonderform des assoziativen Arrays. PHP lässt auch nicht-zusammenhängende Array-Indizes zu, d.h. Sie könnten auch folgendes Array haben:

Schlüssel/Index	Wert
0	Landon
3	Graeme
4	Tobias
10	Till

Bei diesem Array sind nur die Indizes 0, 3, 4 und 10 in Gebrauch, der Rest ist einfach nicht zugeordnet. Mithilfe der Funktion `count()` (welche die Anzahl der zugeordneten Elemente in einem Array zurückgibt) erhalten Sie zwar vier korrekt zugewiesene Elemente, aber Sie können keine `for()`-Anweisung für dieses Array verwenden, da Sie die entsprechenden Schlüssel für die Werte nicht kennen:

```
$my array = array(0 => "Landon", 3 => "Graeme", 4 => "Tobias", 10 =>
➥"Till");

for($i = 0; $i < count($my_array); $i++)
    print("Element $i: $my_array[$i]<br>");
```

Damit erhalten Sie die in Abbildung 2.3 gezeigte Ausgabe.

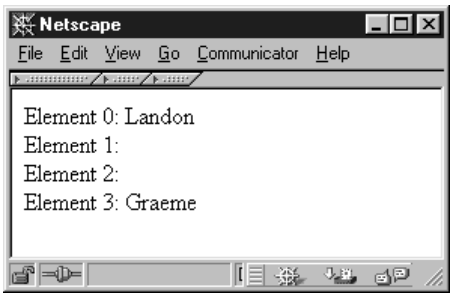

Abbildung 2.3: Ungültiger Zugriff auf ein nicht zusammenhängendes Array

Fehlerberichterstattung
Wenn Sie PHP so eingerichtet haben, dass es ungültige Array-Indizes anzeigt, werden auch ein paar Warnungen ausgegeben. Es empfiehlt sich, während der Entwicklung eine möglichst hohe Ebene für die Fehlerberichte zu wählen.

Eine einfache `for()`-Schleife reicht für Arrays dieser Art nicht aus, denn diese Schleife greift auf Indizes zu, denen noch kein Wert zugewiesen ist. Wenn PHP eine enger definierte Umgebung bereitstellt, würde dies zum Programmabbruch und sofortigen Beendigung des Skripts führen. Wenn Sie den Inhalt und die Konsistenz Ihrer Arrays also nicht genau kennen, sollten Sie `each()` tunlichst nicht verwenden.

`each()` ist auch ein gutes Werkzeug, um sicherzustellen, dass Sie auf Arrays nicht außerhalb ihrer Bereichsgrenzen zugreifen. Auch dies ist ein Grund für Programmunterbrechungen. PHP handhabt Zugriffe außerhalb der Bereichsgrenze auf recht lockere Art (meistens wird nur eine Warnung ausgegeben); trotzdem haben wir es wiederholt geschafft, PHP durch ungültige Array-Zugriffe zum Absturz zu bringen. Im günstigsten Fall reagierte PHP lediglich mit Abbruch des Programms; im schlimmsten Fall belegte das PHP-Modul plötzlich 100% der CPU-Zeit, und der Serverprozess musste abgebrochen werden – eine Situation, die Sie in einer Produktionsumgebung unbedingt vermeiden sollten. In unserem Falle wurde dies wahrscheinlich durch eine fehlerhafte interne Routine zur Array-Bearbeitung in PHP verursacht. Sie sollten trotzdem keine ungültigen Array-Zugriffe provozieren. Es zeugt von schlechtem Programmierstil, und PHP bietet `each()`, `list()` und verwandte Funktionen, um Ihren Programmcode zu sichern. Dies sollten Sie nutzen.

Ursprünglich wurde `each()` zur Benutzung mit »realen« assoziativen Arrays entwickelt, die zur Indexierung von Daten nicht-numerische Schlüssel verwenden. In diesem Fall ist es nicht möglich, auf die gespeicherten Daten ohne eine Funktion zuzugreifen, die alle verfügbaren Schlüssel auflisten kann (sofern Sie nicht wissen, welche Schlüssel in Gebrauch sind). Diese Arrays

könnten auf ähnliche Weise wie das zuvor beschriebene Array organisiert werden, wobei die Schlüssel und Werte in umgekehrter Reihenfolge aufgeführt werden:

```
$my_array = array("Landon" => 1, "Graeme" => 2, "Tobias" => 3, "Till" =>
➥4);

while(list($key, $value) = each($my_array))
    print("Key $key, Value $value<br>");
```

Hier wird das Array nicht über die Nummer, sondern über den Namen indiziert. Wie finden Sie nun heraus, welche Namen in dem Array enthalten sind, wenn Sie keine vordefinierten Schlüssel verwenden? Hier hilft Ihnen `each()`, wie Abbildung 2.4 zeigt.

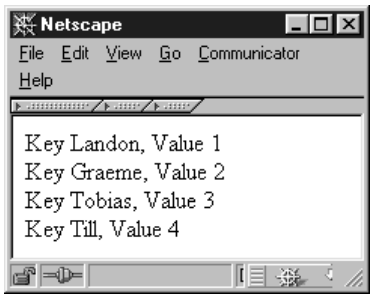

Abbildung 2.4: Assoziative Array-Auflistung mit `each()`

Das Ergebnis ist nicht sehr überraschend, trotzdem sehr nützlich zu wissen.

Ein letzter wichtiger Hinweis zu `each()`: Damit Sie ein Schlüssel/Wert-Paar durch Iteration ermitteln können, muss PHP wissen, auf welches Paar Sie zuletzt zugegriffen haben. Wenn Sie also eine weitere Iteration mit demselben Array durchführen, werden in beiden Fällen nicht dieselben Schlüssel/Wert-Paare zurückgegeben. Um den internen Array-Zähler zurückzusetzen, müssen Sie die Funktion `reset()` verwenden.

Diese Funktion setzt den internen Zeiger von PHP wieder auf das erste Element des Arrays, entsprechend wird dessen Wert als Ausgabewert von `reset()` zurückgegeben.

Das folgende Skript greift in zwei verschiedenen Schleifen zweimal auf dasselbe Array zu, wobei beide Male `each()` verwendet wird:

```
$my_array = array("Landon" => 1, "Graeme" => 2, "Tobias" => 3, "Till" =>
➥4);
print("<h2>Looping without reset()</h2>");
print("<h3>First loop</h3>");
for($i = 0; $i < 2; $i++)
    list($key, $value) = each($my_array);
```

```
    print("Key $key, Value $value<br>");
}
print("<h3>Second loop</h3>");
for($i = 0; $i < 2; $i++)
    list($key, $value) = each($my_array);
    print("Key $key, Value $value<br>");
}
```

Wie die Ausgabe in Abbildung 2.5 zeigt, beginnt die zweite Schleife nicht wieder beim ersten Element. Statt dessen setzt sie dort an, wo die erste Schleife beendet wurde. Dies liegt daran, dass der interne Array-Zeiger von PHP nicht zurückgesetzt wurde. Eine leichte Veränderung des Skripts verändert das Ergebnis (siehe Abbildung 2.6):

```
$my_array = array("Landon" => 1, "Graeme" => 2, "Tobias" => 3, "Till" =>
➥4);
print("<h2>Looping with reset()</h2>");
print("<h3>First loop</h3>");
for($i = 0; $i < 2; $i++)
{
    list($key, $value) = each($my_array);
    print("Key $key, Value $value<br>");
}
print("<h3>Calling reset()</h3>");
reset($my_array);
print("<h3>Second loop</h3>");
for($i = 0; $i < 2; $i++)
{
    list($key, $value) = each($my_array);
    print("Key $key, Value $value<br>");
}
```

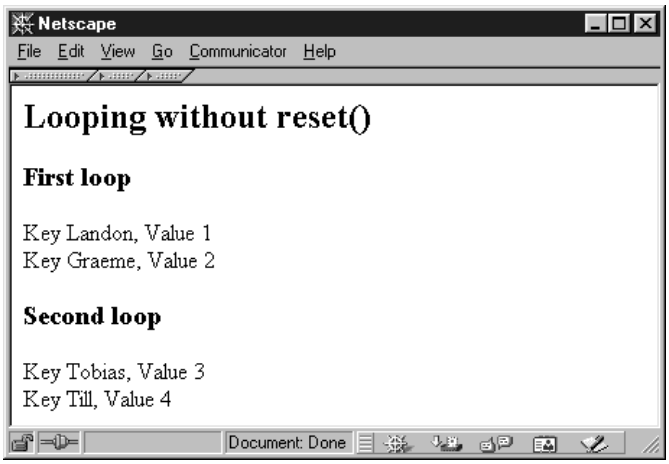

Abbildung 2.5: Verwendung von `each()` in Verbindung mit `reset()`.

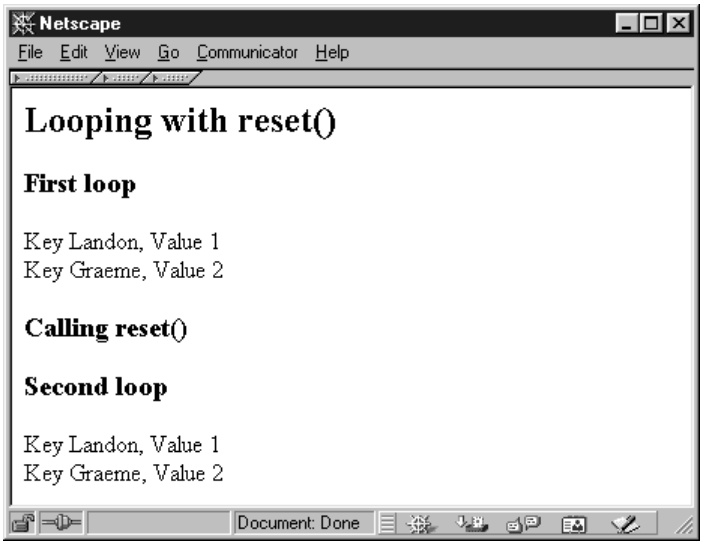

Abbildung 2.6: Verwendung von `each()` in Verbindung mit `reset()`.

Zwischen den beiden Schleifen wurde ein Aufruf von `each()` eingefügt. Die zweite Verwendung von `each()` beginnt beim Lesen des Schlüssel/Wert-Paares aus dem ersten Element.

Neben `reset()` gibt es noch eine Reihe weiterer Funktionen, um die Aktivität von `each()` aufzuteilen: `next()`, `prev()` und `current()`. Mithilfe dieser Funktionen können Sie ein Array manuell in beide Richtungen abarbeiten. `next()` gibt das aktuelle Element zurück und setzt den internen Array-Zeiger auf den nächsten Wert; `prev()` tut das Gleiche, aber setzt den Array-Zeiger auf den vorhergehenden Wert. `current()` gibt nur das Array-Element zurück, auf welchem der Zeiger gerade steht. Diese Funktionen geben jeweils den Wert `false` zurück, wenn sie auf ein leeres oder nicht zugeordnetes Element treffen, oder am Ende des Arrays angelangt sind. Es gibt keine Möglichkeit, diese beiden Fälle zu unterscheiden. Daher sollten diese Funktionen nur verwendet werden, wenn `each()` nicht anwendbar ist und die Möglichkeit, auf ein nichtzugeordnetes Element zu treffen, ausgeschlossen werden kann.

Neben den soeben beschriebenen Funktionen befassen sich noch eine Reihe weiterer PHP-Funktionen mit Arrays. Eine ausführliche Beschreibung finden Sie im Benutzerhandbuch. PHP 4.0 hat eine Vielzahl neuer Array-Funktionen. Auf sie alle einzugehen, würde uns hier zu weit vom eigentlichen Thema abbringen.

2.4 PHP und objektorientierte Programmierung

Anfang der 90er Jahre lernten die meisten bekannten Compiler – wie z. B. die Familie der Borlandcompiler – mit objektorientierten Programmierungserweiterungen (OOP) von »Basissprachen« wie Pascal und C umzugehen. Plötzlich wurden Begriffe wie Klassen, Objekte, Schablonen und Vererbung zu den Schlagworten schlechthin, den heißesten Themen der Softwareentwicklung. OOP war hip und viele Firmen sprangen auf den Zug auf, indem sie ihre Softwarepakete von der prozeduralen auf objektorientierte Programmierung umstellten. Inzwischen ist die Euphorie verblasst, aber eine Sprache, die keine Objekte handhaben kann, wird immer noch als veraltet betrachtet. PHP unterstützt Objekte. Im folgenden Abschnitt wollen wir die Vor- und Nachteile der objektorientierten Programmierung mit PHP erörtern.

Wir halten die Entwicklung zu einer rein objektorientierten Softwareentwicklung für ein wenig zweifelhaft. Große Programmpakete wurden unter erheblichem finanziellem Aufwand in Objekte umgewandelt – nicht eingerechnet der erhebliche Zeitaufwand der Entwickler für die Neukonzeptionierung, Umstrukturierung und Neuinstallation von Tausenden von Programmzeilen. Diese Programmpakete liefen mit prozeduraler Programmierung hervorragend und einige von ihnen benötigten überhaupt keine Objektunterstützung. Wir haben Werbung für Softwaresicherheitssysteme gesehen, die besagte »nun komplett neu implementiert mit objektorientierter Programmierung«. Alles, was diese Systeme taten, war einen Dongle (ein kleines Gerät, das in den Parallelport eines PC eingesteckt wird und als Hardwareschlüssel dient) zu überprüfen, möglicherweise nach einem Passwort zu fragen sowie in einigen Fällen die ausführbaren Programme zu verschlüsseln, mit denen sie verknüpft waren. Die Anwendungen, die sich auf diese Pakete stützten, mussten häufig nicht einmal eine besondere Prozedur starten, um die Passwortüberprüfung einzuleiten. Einige wurden automatisch ausgeführt, sobald die Anwendung gestartet wurde. In anderen Fällen wurde die Überprüfung der Umgebung auf den Aufruf einer einzelnen Funktion beschränkt. Wer braucht jedoch ein Objekt für eine einzelne Funktion? Intern war die Ausführung des Programmcodes streng linear und der Hardwarezugriff wurde von separaten Prozeduren verwaltet (und einige der Anzeigen haben schlichtweg gelogen).

Der fragwürdigste Fall, dem wir begegneten, war ein Entwickler, der an einer Grafikbibliothek arbeitete, um komplexe mathematische zwei- und dreidimensionale Objekte zu zeichnen. Er begründete seine Entscheidung, Objekte zu verwenden, damit, dass er einen Borland-Vertreter auf einer Konferenz getroffen hatte und dieser ihm die objektorientierte Programmierung empfahl. Aber ist es der richtige Weg, die Entwicklung von verlässlichen Softwaresystemen nur auf Empfehlungen zu basieren?

Welche Vorteile haben Objekte nun, und inwiefern unterscheidet sich OOP von dem prozeduralen Ansatz? Warum sollten wir uns überhaupt damit befassen?

Die letzte Frage zuerst. Wir sollten uns gründlich überlegen ob wir in PHP eine objektorientierte Programmierung einsetzen, denn es macht wenig Sinn, eine Technik zu verwenden, die eine größere Belastung für die Entwicklung darstellt, von der zugrunde liegenden Architektur nicht ausreichend unterstützt wird und für die Anwendung im Grunde genommen keinen Unterschied macht. Prozedurbasierte Projekte können genauso effektiv, leicht zu pflegen und erweiterbar sein wie objektorientierte Projekte. Tabelle 2.1 zeigt die wichtigsten Unterschiede zwischen den beiden Ansätzen.

Objekte	**Prozeduren**
Vollständige Verschachtelung von Daten	Keine Verschachtelung von Daten; nur über separate Parameter
Mehrere Instanzen sind zugelassen	Mehrere Instanzen nicht erlaubt; unterschiedliche Datensätze müssen durch Kopien der vorhandenen Variablen verwaltet werden
Ermöglicht zusätzliche Funktion über Vererbung bei Erhalt der Schnittstelle	Keine Vererbung; zusätzliche Funktionalität nur über die API durch Bereitstellung einer weiteren API-Schicht oder durch Ändern der gesamten API
Selbstzentriert; das Objekt hat seinen eigenen Datensatz und muss nur für dessen Gültigkeit sorgen und den Zugriff für andere Parteien gewährleisten	Global ausgerichtet; Prozeduren haben keine eigenen Datensätze. Die Daten werden vom Anrufer bereitgestellt.
Bietet einfache Mittel zur Sicherstellung der Datenintegrität	Initialisierung und Cleanup (Konstruktor/Destruktor) Datenintegrität kann nur schwer sichergestellt werden. Initialisierung und Cleanup müssen explizit aufgerufen werden.
Isolierter Namensraum	Namen müssen in den globalen Namensraum eingeführt werden

Tab. 2.1: Objektorientierte Programmierung versus prozedurale Programmierung

Diese Tabelle führt nur die wichtigsten Unterschiede auf. Es gibt noch weitere, aber wir können bereits sehen, dass es für Prozeduren nicht besonders gut aussieht. Sind Prozeduren wirklich so schlecht, wie es den Anschein hat? Bedeutet dies, dass die objektorientierte Programmierung die prozedurale Programmierung komplett ablöst? Tatsächlich hängt dies von Ihren Zielen und der Plattform ab, auf der Sie arbeiten. In unserem Fall ist die Plattform PHP. PHP unterstützt natürlich Objekte, jedoch auf eine sehr spezielle Weise. Dies hängt damit zusammen, wie der Interpreter Variablen handhabt.

Sobald PHP auf eine Anweisung trifft, für die ein Schreibzugriff auf eine Variable benötigt wird, wertet es die Daten aus, die in die Variable geschrieben werden, berechnet sie, kopiert diese und weist das Ergebnis dem Zielspeicherbereich zu. (Diese Beschreibung ist zwar etwas vereinfacht, vermittelt Ihnen aber das Grundprinzip.)

```
$some_var = 2;

$my_var = $some_var * 3;

$new_var = $some_var + $my_var;
```

Laut diesem Skript wird PHP

- einen Speicherbereich für `$some_var` erzeugen und »2« hineinschreiben.
- einen Speicherbereich für `$my_var` erzeugen, den Wert von `$some_var` auslesen, ihn mit 3 multiplizieren und das Ergebnis dem neu angelegten Speicherbereich zuordnen.
- einen Speicherbereich für `$new_var` erzeugen, die Werte von `$some_var` und `$my_var` auslesen, sie zusammenzählen und sie wieder in den neuen Speicherbereich zurückschreiben.

Dies klingt alles sehr schön und logisch – es handelt sich jedoch um einfache Typen, mit denen Sie bereits viele Male gearbeitet haben. Dies sieht ganz anders (und unlogisch) aus, sobald PHP mit Klassen zu tun hat:

```
class my_class
{
    var $var1, $var2, $var3;
}

$my_object = new my_class;

$my_object->var1 = 1;
$my_object->var2 = 2;
$my_object->var3 = 3;

$new_object = $my_object;

$new_object->var1 = 3;
$new_object->var2 = 2;
$new_object->var3 = 1;

print("My object goes $my_object->var1, $my_object->var2,
➥$my_object->var3 !<br>");
print("New object goes $new_object->var1, $new_object->var2,
➥$new_object->var3 !<br>");
```

Was glauben Sie, wird hier als Ergebnis herauskommen? Das Skript deklariert zunächst eine Klasse, erzeugt eine Instanz davon und weist seinen drei Elementen Werte zu. Anschließend erzeugt es eine neue Referenz auf dieses Objekt, weist seine Elemente neu zu und gibt jedes Element aus, wobei es beide Referenzen verwendet. Bedenken Sie, es handelt sich um *eine* Instanz. Abbildung 2.7 zeigt das Ergebnis.

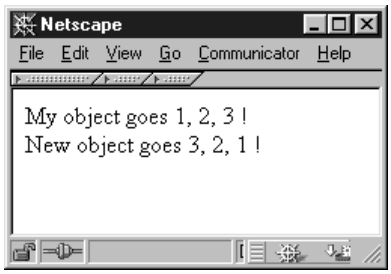

Abbildung 2.7: Erzeugen einer Kopie anstelle einer Referenz mit PHP

Wenn Sie jetzt nicht überrascht sind, kennen Sie PHP bereits recht gut oder haben noch nicht intensiv genug über Objekte nachgedacht. PHP hat nicht einfach eine neue Referenz von `my_class` erzeugt sondern eine Kopie. Dies ist nicht das gewünschte Verhalten, da der Operator `new` eigentlich im Speicherbereich eine Instanz von `my_class` erzeugen und einen Verweis auf diesen zurückgeben sollte. Wenn Sie diese Referenz einer anderen Variablen zuordnen, sollte damit nur die Referenz kopiert werden und die Originaldaten unberührt bleiben, vergleichbar mit einer Dateisystemverknüpfung, die einen Zugriff auf dieselben Daten von verschiedenen Speicherorten im Dateisystem ermöglicht. Dieses Verhalten von PHP – d.h. anstelle der Referenz Kopien referenzierter Daten zu erzeugen – klingt nicht so bedeutend, als dass wir uns darauf konzentrieren sollten. Sie werden jedoch in Kürze feststellen, dass es durchaus von Bedeutung ist.

Hinweis: Zum Zeitpunkt, da dieses Handbuch erstellt wurde, verwendeten sowohl PHP 3.0 als auch PHP 4.0 die Kopiersyntax. Ein Gespräch mit einem Mitglied aus dem Kernentwicklungsteam ergab, dass dieses Standardverhalten geändert werden und künftig eine Referenzsyntax verwendet werden soll. Diese Änderung würde jedoch zum Verlust der Rückwärtskompatibilät führen. Die Änderung ist für die Version 4.1 geplant. Danach wären alle Informationen hier für künftige Versionen nicht mehr gültig.

Warnung: PHP 3.0 verfügt über keinen vernünftigen Mechanismus zur Speicherbereinigung. Sobald Sie einen Wert in eine Variable schreiben, wird nicht der alte Speicherbereich wiederverwendet, sondern neuer Speicher zugewiesen. Mit `unset()` können Sie dies in begrenztem Maße umgehen – es wird zwar kein Speicher freigegeben, sondern nur als wiederverwendbar markiert – auf lange Sicht wird der Speicher Ihres Servers jedoch insbesondere durch langfristige Projekte belegt. Wenn Sie langfristige Skripte verwenden wollen, dann stellen Sie sicher, dass Sie Ihre Datenbankergebnisse beispielsweise mit `mysql_free_result()` freigeben und `unset()` auf alle Variablen anwenden, die keine wertvollen Informationen mehr enthalten. Der Speicher wird jedoch erst frei, wenn das Skript beendet ist!

Nehmen wir beispielsweise eine Baumstruktur. Die Klasse zum Aufbau der Baumknoten sieht folgendermaßen aus:

```
class tree_node
{
    var $left_child, $right_child;
    var $value;
}
```

Dies ist zwar nur ein einfacher Baumknoten, aber er enthält alles, was wir benötigen: eine Verknüpfung zum linken Tochterknoten und eine Verknüpfung zum rechten Tochterknoten sowie eine Variable, die den Inhalt dieses Knotens enthält. Lassen Sie uns nun einen einfachen Baum konstruieren:

```
$root_node = new tree_node;
$left_node = new tree_node;
$right_node = new right_node;

$root_node->value = 1;
$left_node->value = 2;
$right_node->value = 3;

$root_node->left_child = $left_node;
$root_node->right_child = $right_node;
```

Dieser Programmcode erzeugt einen Baum mit einem Stammknoten und zwei Tochterknoten, wobei jedem dieser Knoten ein anderer Wert zugewiesen wird. Die Abarbeitung des Baums könnte mit einer Funktion wie dieser erfolgen:

```
function traverse_tree($start_node)
{

    $node = $start_node;
    print("Value is $node->value<br>");
```

```
    print("Traversing left tree:<br>");
    traverse_tree($node->left);

    print("Traversing right tree:<br>");
    traverse_tree($node->right);

}
```

Lassen wir für einen Moment außer acht, dass diese rekursive Funktion nie enden wird, da unser Baum keine Stopp-Markierung besitzt (die Funktion weiß nicht, welcher Knoten tatsächlich einen Tochterknoten besitzt und welcher nicht). Sehen wir uns seine Funktionsweise an und wie PHP damit umgeht.

Der kritische Punkt ist bereits die erste Zeile, in der `$node` ein Wert zugewiesen wird. Der Parameter `$start_node` gibt die Instanz des Knotens an, von welcher gestartet werden soll, und die Zuordnung zu `$node` erzeugt eine Kopie davon. Für eine Funktion, die den Baum einfach rekursiv abarbeitet und den Knoteninhalt zurückgibt, ist die Tatsache, dass eine Kopie erzeugt wird, nicht von Bedeutung; es kann jedoch wichtig werden, wenn Sie den Baum an einer Stelle ändern möchten.

Nehmen wir nun an, Sie möchten eine Funktion schreiben, die an den Knoten ganz links außen einen weiteren Knoten anhängt. Kein Problem. Schreiben Sie einfach eine rekursive Funktion, welche die Sprünge nach links zählt und den Knoten mit der höchsten Zählerzahl zurückgibt. Ändern Sie dann die Verknüpfung zu den Tochterknoten dieses Objekts und Sie sind fertig. Moment – sind Sie wirklich fertig? Denken Sie einen Moment darüber nach, was Sie geändert haben (siehe Abbildung 2.8). Sie haben die Kopie des Knotens links außen geändert, nicht jedoch den Knoten selbst! Sobald Sie diese Funktion verlassen, gehen Ihre Änderungen unwiderruflich verloren.

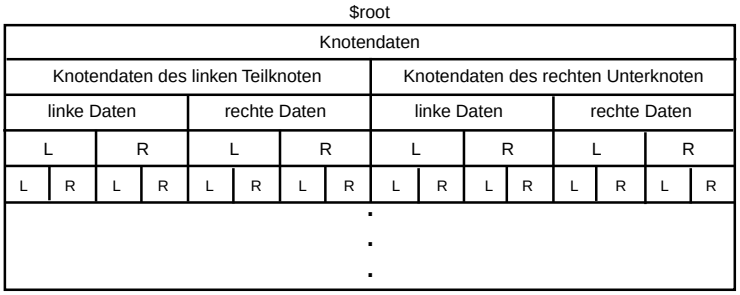

Abbildung 2.8: Baumstruktur, erzeugt mithilfe der Kopiersyntax

Sie könnten auch eine Referenz auf diese Instanz zurückgeben und diese ändern, indem Sie den einfachen »Zeiger«-Mechanismus verwenden, den PHP bereitstellt. Dann haben Sie das Objekt selbst geändert und nicht seine

Kopie. Sobald Sie den Baum wieder abarbeiten, werden Sie jedoch etwas Merkwürdiges feststellen: Nichts scheint sich geändert zu haben. Tatsächlich hat sich nichts geändert. Dies liegt daran, dass der Elternknoten des Knotens, den Sie ändern wollten, eine eigene Kopie derselben Daten besitzt! Bedenken Sie, dass `$node->left` nichts Anderes ist als eine weitere Kopie eines Knotens im Baum. Dies ist genau die Kopie, welche die Bearbeitungsroutine auswertet. Die Kopie, die Sie geändert haben, bleibt außerhalb des Baumes und wird schließlich bei einer Speicherbereinigung gelöscht. Um auch diesen Knoten zu ändern, müssen Sie Referenzen an die Elternknoten, die Großelternknoten und die Ur-Großelternknoten weiterleiten. Damit landen Sie bei einer weiteren rekursiven Funktion, die alles Andere als einfach zu programmieren ist und in 99% aller Fälle nicht funktionieren wird.

2.4.1 Klassen: PHP 3.0 versus PHP 4.0

PHP 4.0 hat aus der Unfähigkeit von PHP 3.0 gelernt, Objektreferenzen zu verwalten, und bietet nun »echte« Referenzen. »Echt« ist hier in Anführungszeichen gesetzt, da diese nicht wirklich auf den Speicherbereich zeigen, den die andere Variable belegt. PHP interpretiert diese Variablen nur als Referenzen und verhält sich anders. Eingebettet in das vorangegangene Beispiel, sähe der Programmcode folgendermaßen aus:

```
// create multiple references to the original object
$new_object = &$my_object;
$another_object = &$new_object;
```

Dieser Code erzeugt zwei Referenzen auf dasselbe Objekt. Beachten Sie, dass `$another_object` als Referenz `$new_object` zugeordnet wird, nicht als Kopie hiervon. Wenn Sie versuchen die Referenzen zu kopieren, kopiert PHP nicht die Referenz, sondern erzeugt statt dessen eine Kopie der referenzierten Variablen. Nur wenn Sie jetzt Referenzen verwenden (um eine Referenz zu kopieren, müssen Sie wieder den Referenzoperator @ verwenden), können Sie sowohl `$new_object` als auch `$another_object` verwenden, um die Daten zu ändern, die sich innerhalb von `$my_object` befinden.

Für die unterschiedlichen Versionen geben wir einige Empfehlungen, auf die wir nachfolgend eingehen.

Klassen in PHP 3.0

Daten: Verwenden Sie Klassen nicht für komplexe Datenstrukturen, die echte Zeiger benötigen (wie etwa Bäume). Wenn es sich nicht vermeiden lässt, versuchen Sie die Verwendung von Klassen auf die Erfassung von Daten zu beschränken und die Datenverwaltung auszusparen.

Code: Verwenden Sie Klassen nur zur Strukturierung von APIs, die nicht durch die Kopiersyntax verletzt werden. Überlegen Sie, ob sich Ihr Projekt mit Prozeduren realisieren lässt. Wenn ja, erforschen Sie diese Möglichkeit. Prozeduren sind zuverlässig und haben sich bewährt, wohingegen Objekte noch einige Gefahren bergen.

Klassen in PHP 4.0

Verwenden Sie Klassen vorsichtig und stellen Sie sicher, dass Sie Kopien und Verweise von Objekten unterscheiden können. Achten Sie darauf, an welche Typen Sie Werte übergeben und wie Sie mit Ihnen umgehen. Sobald Sie nur ein einziges @ vergessen, erzeugt PHP eine Kopie Ihres Objekts, was wahrscheinlicher Weise die Konsistenz Ihrer Daten unterbrechen wird.

Die Klassen wurden zwar in PHP 4.0 verbessert, aber wir bleiben skeptisch. Wir haben Meinungen gehört, die in beide Richtungen gehen. Eine Gruppe besteht darauf, dass Objekte nichts taugen und nicht in einer Sprache wie PHP verwendet werden sollten, während eine andere Gruppe Objekte jedem prozeduralen Ansatz vorzieht, sogar in PHP 3.0. Für jene, die nur mit Prozeduren arbeiten, scheinen Objekte ein schlafendes Ungeheuer zu sein, das man besser nicht wecken sollte. Für die Fangemeinde der *objektorientierten Programmierung* sind die Anhänger des prozeduralen Ansatzes Idioten. Es ist fast wie ein Religionskrieg. Wann immer wir das Thema anschnitten, führte es zu endlosen und fruchtlosen Diskussionen.

Nach unserer Ansicht sind beide Extreme falsch. Es ist weder gut Funktionen komplett zu ignorieren noch sie ohne Betrachtung der Nachteile zu verwenden. Uns widerstrebt es aber auch, es als persönliche Vorliebe abzutun. Technologien sollten nie unter diesem Gesichtspunkt betrachtet werden. Unsere Empfehlung: Befreien Sie sich von jeglichen Vorurteilen, die Sie möglicherweise haben, insbesondere denjenigen von anderen, und entscheiden Sie dann objektiv, was für Ihr Projekt am besten geeignet ist.

2.4.2 Implementierung von Klassen

Ungeachtet der Pros und Contras sind Klassen ein wichtiges Element der Programmiersprache, und es scheint immer einen Bedarf zu geben, die Implementierung von Klassen in PHP zu erläutern.

Tatsächlich lassen sich Klassen in PHP recht einfach implementieren. Die meisten Schlüsselworte werden Sie bereits aus anderen Sprachen kennen:

```
class shopping_cart
{
    var $item_list;

    function pick($item, $quantity)
```

```
    {

        $this->item_list[$item] += $quantity;

    }

    function drop($item, $quantity)
    {

        if($this->item_list[$item] > $quantity)
            $this->item_list[$item] -= $quantity;
        else
            $this->item_list[$item] = 0;

    }

}
```

Dieser Code definiert die Klasse `shopping_cart` mit den Mitgliedern `$item_list`, `pick()` und `drop()`. Es gibt keine Möglichkeit öffentliche und private Mitglieder zu unterscheiden. In PHP wird standardmäßig alles auf öffentlich gesetzt – was bedeutet, dass Sie uneingeschränkt auf alle Elemente und Mitgliedsfunktionen einer Klasse von außen zugreifen können.

Dieses einfache Klassenbeispiel implementiert eine Art von Einkaufswagen (wie der Name sagt) mit einer Variablen, die den Inhalt des Wagens in einem assoziativen Array (`$item_list`) angibt, sowie zwei Funktionen (`pick()` und `drop()`), mit denen Einträge hinzugefügt und entfernt werden. Die Mitgliedsfunktionen sind wie reguläre Funktionen deklariert, mit der Ausnahme, dass sie in der Klassendefinition eingebettet sind. Die Klassenelemente (Variablen innerhalb der Klassen) werden mithilfe des Schlüsselworts »var« definiert.

Hinweis: In PHP können Sie Klassendeklarationen und Implementierungen nicht trennen. Alle Funktionen müssen direkt in der Klassendeklaration implementiert werden.

2.4.3 Zugriff auf Objekte

Die Mitgliedsfunktionen können entweder über die »alte« Syntax `instance->member()` oder die »neue« Syntax `instance::member()` aufgerufen werden. Auf die gleiche Weise greifen Sie auf die Elemente entweder mit `instance->property` oder mit `instance::property` zu. Letztere Form eignet sich insbesondere dazu, den Konstruktor eines Elternknotens aufzurufen oder auf andere Mitglieder zuzugreifen, die sich nicht im aktuellen Objekt befinden (mehr zu diesem Thema im Abschnitt »Vererbung«):

```
class extended_cart extends shopping_cart
{

    function extended_cart()
    {

        shopping_cart::shopping_cart("Mousepad", 1);

    }

    function query($item)
    {

        return($this->item_list[$item]);

    }

}
```

Diese erweiterte Version des Objekts `extended_cart` besitzt einen Konstruktor, der den Konstruktor des Elternknotens aufruft, damit dieser den Rest den Objektbaums korrekt initialisiert. Würde der Konstruktor des Elternknotens nicht explizit von diesem Konstruktor aufgerufen, würde der Elternknoten nicht initialisiert werden (und damit auch nicht der Elternknoten dieses Knotens usw.).

Außerdem besitzt PHP einen Alias, der auf die aktuelle Instanz eines Objekts zeigt. Dieser Alias heißt `this` und gewährt Zugriff auf alle Mitglieder der aktuellen Instanz. Er wird für alle Selbstbezüge benötigt, da PHP innerhalb von Klassendefinitionen keinen lokalen Gültigkeitsbereich festlegt.

Hinweis: Da PHP keinen lokalen Gültigkeitsbereich innerhalb der Klassendefinitionen festlegt, stellen Sie sicher, dass Sie das Schlüsselwort `this` immer dann verwenden, wenn Sie innerhalb Ihres Objekts referenzieren. Wenn Sie `this` vergessen, referenziert PHP automatisch auf den globalen Gültigkeitsbereich, was recht fehlerträchtig ist!

2.4.4 Konstruktoren

Konstruktoren sind wie reguläre Funktionen definiert, mit der Ausnahme, dass ihre Namen mit den Klassennamen identisch sein müssen. Destruktoren kennt PHP nicht. Wie jede andere Funktion können auch Konstruktoren Argumente besitzen, selbst optionale Argumente sind zugelassen (weitere Informationen zu optionalen Parametern finden Sie weiter unten im Abschnitt über variable Argumentenlisten).

Hinweis: Seit PHP 4.0 sind für Kostruktoren nur skalare Werte als Parameter zugelassen (Zeichenketten, ganze Zahlen usw.), jedoch keine Arrays oder Objekte, wie es noch in PHP 3.0 der Fall war.

Um im vorangegangenen Beispiel einen Konstruktor hinzuzufügen, erweitern wir den Programmcode wie folgt:

```
class shopping_cart
{
    var $item_list;

    function shopping_cart($item = "T-Shirt", $quantity = 1)
    {

        $this->pick($item, $quantity);

    }

    function pick($item, $quantity)
    {

        $this->item_list[$item] += $quantity;

    }

    function drop($item, $quantity)
    {

        if($this->item_list[$item] > $quantity)
            $this->item_list[$item] -= $quantity;
        else
            $this->item_list[$item] = 0;

    }

}
```

Der Konstruktor, der in `shopping_cart::shopping_cart()` enthalten ist, hat zwei optionale Argumente. Wenn keine Argumente angegeben sind, wird der Einkaufswagen automatisch mit einem T-Shirt »vorgefüllt«. Ansonsten erhält er die gewünschten Einträge:

```
$default_cart = new shopping_cart;        // this cart will fill itself
                                             ➥ with one T-Shirt by default
$mug_cart = new shopping_cart("Mug", 2);  // this cart will contain two
                                             ➥ mugs
```

2.4.5 Vererbung

Um Funktionen zu Objekten hinzuzufügen sollte nicht der alte Code neu geschrieben, sondern existierende Strukturen stärker ausgenutzt werden. Neue Objekte können mithilfe des Schlüsselworts `extends` aus alten Objekten übernommen werden. Wie der Name sagt, wird hierdurch eine neue Klasse als Erweiterung einer bestehenden definiert:

```
class extended_cart extends shopping_cart
{

    function query($item)
    {

        return($this->item_list[$item]);

    }

}
```

Dieser erweiterte Wagen `extended_cart` enthält nun alle Elemente und Mitgliedsfunktionen von `shopping_cart` sowie zusätzlich eine weitere Funktion `query()`, mit welcher wir die Menge eines bestimmten Eintrags im Wagen überprüfen können.

Hinweis: Die Klasse `extended_cart` besitzt keinen eigenen Konstruktor. Wenn die Klasse eines Tochterknotens keinen Konstruktor hat, ruft PHP (seit der Version 4.0) automatisch den Konstruktor des Elternknotens auf. Standardmäßig wird PHP jedoch nie den Konstruktor eines Elternknotens aufrufen. Wenn Sie also das übergeordnete Objekt einrichten müssen, stellen Sie sicher, dass Sie seinen Konstruktor manuell aufrufen.

2.4.6 Spezielle Funktionen für objektorientierte Programmierung

PHP hat ein paar schöne Funktionen, welche den Umgang mit Objekten erleichtern. Die folgende Tabelle beschreibt diese Funktionen.

Funktion	Beschreibung
`string get_class (object object)`	Gibt den Namen der angegebenen Objektinstanz als Zeichenkette zurück.
`string get_parent_class (object object)`	Gibt den Namen der übergeordneten Klasse der angegebenen Objektinstanz als String zurück.

Funktion	Beschreibung
`bool method_exists (object object, string method)`	Prüft, ob die in `method` genannte Funktion tatsächlich ein Mitglied von `object` ist.
`bool class_exists (string classname)`	Prüft, ob `classname` eine tatsächlich definierte Klasse ist.
`bool is_subclass_of (object object, string classname)`	Prüft, ob `object` eine Unterklasse von `classname` ist.

Hinweis: Diese Funktionen sind nicht in PHP 3.0 verfügbar.

Quellcode für den »Einkaufswagen«

Dieser Abschnitt erläutert anhand des »Einkaufswagens« die vollständige Implementierung eines objektorientierten Programms (siehe Listing 2.1). Das Beispiel ist zwar sehr einfach, aber trotzdem nützlich.

```
class shopping_cart
{

    var $item_list;

    function shopping_cart($item = "T-Shirt", $quantity = 1)
    {

        $this->pick($item, $quantity);

    }

    function pick($item, $quantity)
    {

        $this->item_list[$item] += $quantity;

    }

    function drop($item, $quantity)
    {

        if($this->item_list[$item] > $quantity)
            $this->item_list[$item] -= $quantity;
        else
            $this->item_list[$item] = 0;

    }
```

```
}

class extended_cart extends shopping_cart
{

    function query($item)
    {

        return($this->item_list[$item]);

    }

    function get_contents()
    {

        return($this->item_list);

    }

}

// you can instantiate shopping_cart the regular way
$cart = new shopping_cart;

// you can make use of the variable arguments of the constructor
$cart = new shopping_cart("Cap", 2);

// you can also use extended_cart, which in turn calls the
// constructor of shopping_cart implicitly
$cart = new extended_cart;

// or you can use the inherited features of the constructor
$cart = new extended_cart("Cap", 2);

// of course, you can also use the inherited functions
$cart->pick("Mug", 1);

// ...or use any functions in the object itself
while(list($item, $quantity) = each($cart->get_contents()))
    print("We have $quantity of $item");
```

Listing 2.1: Einkaufswagen-Quellcode

2.5 Verknüpfte Listen

Verknüpfte Listen, eine Sonderform der Baumstrukturen, gehören zu den typischen Datenstrukturen für die Organisation von dynamischen Datensätzen. Wir gehen davon aus, dass Sie bereits ein Grundverständnis von der Struktur, dem Konzept und der Verwendung verknüpfter Listen haben. Daher werden wir hier nicht detailliert auf ihre Implementierung eingehen, sondern konzentrieren uns auf die wichtigsten zu beachtenden Aspekte.

Wie bereits in den vorangegangenen Abschnitten beschrieben, erzeugt PHP 3.0. von den Objektinstanzen Kopien, anstatt diese mithilfe eines Zeigers zu referenzieren. Damit ist nur eine sehr begrenzte, WORM-ähnliche Verwendung der verknüpften Listen möglich: einmal schreiben, mehrfach lesen. Verknüpfte Listen können erzeugt, aber nicht verändert werden. Wenn Sie versuchen, ein Element in der Liste zu ändern, verlieren Sie die Referenz auf alle folgenden Elemente in der Liste. Aus demselben Grund lässt sich auch die Reihenfolge der Elemente nicht verändern.

Doppelt verknüpfte Listen lassen sich in PHP 3.0 ebenfalls nicht realisieren (zumindest haben wir es nicht geschafft, wobei wir einige Stunden mit Fehlersuche und Behebung zubrachten, bevor wir aufgegeben haben). Da jeder Knoten eine neue Kopie des Listenendes benötigte, mit dem es verknüpft ist, müssten Sie eine Vielzahl redundanter Listen mit demselben Inhalt erstellen, nur um die Funktion »ein Element zurückgehen« zu ermöglichen.

PHP 4.0, das echte Referenzen unterstützt, hat diese Einschränkungen nicht. Listen können beliebig erzeugt und neu zusammengestellt werden, selbst doppelt verknüpfte Listen. Beachten Sie jedoch, dass es nicht ganz einfach ist, zwischen Referenzen und den tatsächlichen Kopien von Listenelementen zu unterscheiden.

Ein Motto von Programmierern konventioneller Programmiersprachen lautet: Hüte dich vor freien Zeigern. Wir möchten dieses für PHP modifizieren: Hüte dich vor redundanten Kopien.

Wenn Sie mit Listen arbeiten, erstellen Sie eine absturzsichere Bibliothek, die auf möglichst allgemeine Weise Ihre Daten nach Ihren Bedürfnissen verwaltet. Testen Sie die Bibliothek ausführlich und stellen Sie sicher, dass sie korrekt arbeitet. Das erspart Ihnen, fehlerhaften Programmcode zu suchen, der auf Ihre Listen auf falsche Weise zugreift und diese möglicherweise zerstört.

2.5.1 Verknüpfte Listen und Baumstrukturen – ein Arbeitsansatz

Wie bereits erwähnt, empfiehlt es sich, für Ihre Bedürfnisse eine absturzsichere Bibliothek zu erstellen – eine, die sich einfach erweitern lässt und alle für die Aufgabe erforderlichen Informationen bereitstellt. Im Folgenden

möchten wir ein Beispiel aus der Praxis vorstellen: eine Bibliothek, die wir zur Verwaltung von Baumstrukturen entwickelt haben, und die auch mit PHP 3.0 arbeitet. Den vollständigen Quellcode finden Sie auf der CD-ROM.

Die Bibliothek kann doppelt verknüpfte Baumstrukturen mit zwei Tochterknoten pro Blattknoten handhaben, wobei jeder Knoten einen Inhaltsbehälter für gemischte Variablen hat. Jede Aktion, die sich mit der Baumstruktur durchführen lässt, wurde in die Anwendungsprogrammierschnittstelle eingebaut. Sie erhalten zwei separate Einheiten: die Baumstruktur und den auf sie zugreifenden Programmcode.

Dies ist genau der Grund, warum diese Baumstruktur mit PHP 3.0 funktioniert (auch wenn dies scheinbar in Widerspruch zu dem zuvor Gesagten steht). Die Baumstruktur basiert nicht auf Zeigern, sondern auf Arrays. Da PHP dynamische Arrays unterstützt, ist es mit einem geringen Aufwand möglich, diese Arrays zu Erstellung einer dynamischen Baumstruktur zu verwenden.

Die Idee ist nicht neu. Sie existiert bereits seit einigen Jahren und ist nicht schwer zu verstehen. Anstatt Zeiger zu verwenden, welche auf die Speicheradresse eines anderen Knotens verweisen, mit welchem ein Knoten verknüpft ist, sind im Array alle Knoten mit den Indizes der Knoten versehen, mit denen sie verknüpft sind. Dies hat nicht nur den Vorteil, dass PHP bei Verwendung von ungültigen Indizes eine Warnung ausgibt, Sie können auch den gesamten Baum kopieren, indem Sie die Variable, die das Array eines Baumes angibt, einer anderen Variablen zuweisen. Darüber hinaus können Sie die gesamte Baumstruktur, so wie sie ist, in eine Datei kopieren und diese an eine beliebige andere Stelle verschieben.

Eine weitere, eher theoretische Erläuterung, ist folgende: Stellen Sie sich den für Ihr Programm verfügbaren Speicher als ein großes Array vor. Die Elemente wären im Falle von physischem RAM wahrscheinlich einige Byte groß, aber die Größe der einzelnen Elemente ist nicht wirklich von Bedeutung. Wichtiger ist, dass der Zeiger einfach eine Nummer ist, die eines der Elemente indiziert und damit den Beginn jeder Struktur markiert, die Sie in Ihr RAM einfügen. Wenn Sie nun das gesamte Konstrukt in eine Sprache integrieren (ein »echtes« Array), haben Sie dieselbe Situation auf einer höheren Ebene. Das PHP-Array enthält nur Ihr »RAM«, und jedes Array-Element stellt einen der drei Baumknoten dar. Die Zeiger werden in diesem Array zu Indizes, und die Referenzierung erfolgt einfach durch Auslesen des korrekten Elements aus dem Array.

Mithilfe von Arrays können Sie viele »RAMs« erzeugen, ihre Größe erhöhen oder verringern, über sie als Ganzes oder nur ein einzelnes Element verfügen; alles in allem eine sehr komfortable Methode zur Speicherverwaltung. Wenn Sie dies alles in einer soliden Bibliothek integriert haben, besitzen Sie ein schönes Werkzeug.

Abbildung 2.9 zeigt, wie die Baumbibliothek die Baumknoten in einem Array intern verwaltet.

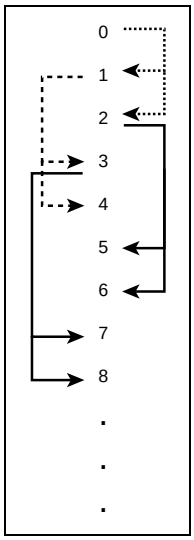

Abbildung 2.9: Array, das eine Baumstruktur enthält

Die Bibliothek besteht aus den folgenden Funktionen:

Funktion	Beschreibung
`array tree_create()`	Erzeugt einen neuen Baum.
`int tree_allocate_node (array tree)`	Ordnet in der Baumstruktur einen neuen Knoten zu.
`int tree_free_node (array tree, int handle)`	Gibt einen Knoten in der Baumstruktur frei.
`int tree_link_left (array tree, int link_to, int child)`	Verknüpft einen Knoten als linken Tochterknoten mit einem anderen Knoten.
`int tree_link_right (array tree, int link_to, int child)`	Verknüpft einen Knoten als rechten Tochterknoten mit einem anderen Knoten.
`int tree_get_parent (array tree, int handle)`	Gibt den Elternknoten des angegebenen Knoten zurück.
`int tree_get_left (array tree, int handle)`	Gibt den linken Tochterknoten des angegebenen Knotens zurück.
`int tree_get_right (array tree, int handle)`	Gibt den rechten Tochterknoten des angegebenen Knotens zurück.

Funktion	Beschreibung
`int tree_assign_node_contents (array tree, int handle, mixed contents)`	Weist dem Inhaltsbehälter eines Knotens Daten zu.
`mixed tree_retrieve_node_contents (array tree, int handle)`	Liest Daten aus dem Inhaltsbehälter eines Knotens aus.

Wenn Sie möchten, können Sie hier weitere Funktionen hinzufügen. Die Bibliothek hat beispielsweise noch keine Funktionen um Bäume zu vermischen, tote Knoten zu entdecken (Knoten, die zwar im Array vorhanden, aber vom Hauptbaum abgeschnitten sind und auf die somit nicht mehr zugegriffen werden kann) usw. Dieser Code eignet sich hervorragend zum Experimentieren und Lernen.

Die Bibliothek ist recht einfach. `tree_create()` erzeugt ein neues Array für die Baumstruktur und initialisiert das erste Element als Stammelement. Alle Referenzen auf andere Knoten sind ganzzahlige Indizes innerhalb des Arrays (siehe `$idx_up`, `$idx_left` und `$idx_right` in der Quelle). -1 markiert eine gerade verwendete Referenz. Wenn ein Knoten beispielsweise keinen linken Tochterknoten hat, enthielte `$idx_left` die Angabe -1. Um zu markieren, ob ein Element selbst in Gebrauch ist oder nicht (d.h. ob ihm Daten zugewiesen sind), wird ein anderer Flag definiert: `$free`. Diese Variable hat entweder den Wert `1` (in Gebrauch) oder `0` (nicht in Gebrauch).

`tree_create()` erzeugt einen Dummyknoten und markiert ihn als frei, d.h. alle Referenzen werden nicht verwendet, und weist ihn dem Slot 0 in dem Baum-Array zu. Anschließend wird dieses Array an den Anrufer zurückgeschickt.

Hinweis: Der Anrufer muss nichts über dieses Array wissen, nicht einmal dass es ein Array ist. Das Programm sollte es einfach für etwas als Baumstruktur »zu bearbeitendes« halten. Da PHP keine expliziten Typen hat und verlangt, funktioniert dies sehr gut.

`tree_allocate_node()` sucht innerhalb des Baum-Arrays nach einem freien Knoten, indem es für jeden bestehenden Knoten den Flag `$free` überprüft. Wenn keiner der Knoten als frei markiert ist, weist es einfach einen neuen zu und fügt es dem Baum-Array hinzu. Hier macht sich die dynamische Struktur von PHP bemerkbar: Wenn wir Arrays fester Größe nutzen müssten, gingen uns früher oder später die Knoten aus. Der gefundene Knoten wird dann als »in Gebrauch« markiert und als »zu bearbeiten« an den Anrufer zurückgeschickt.

`tree_free_node()` macht genau das Gegenteil. Es markiert den angegebenen Knoten als »nicht in Gebrauch«, indem es den Flag `$free` löscht. Dies wirft drei Probleme auf: Zunächst sind freie Knoten nicht wirklich frei, sie sind nur als frei markiert. Angenommen, Sie wollten eine komplexe Baumstruktur mit vielen Knoten erstellen und anschließend ein Optimierungsprogramm darüber laufen lassen, das in der Regel die Anzahl der Knoten um die Hälfte reduziert und viele Geisterknoten im Array zurücklässt. Wenn wir davon ausgehen, dass Sie ursprünglich 1.000 Knoten zugeordnet haben und während der Optimierung 500 von ihnen freisetzen, erhalten Sie ein Array, das immer noch 1.000 Knoten enthält, wobei hierbei nur 500 von ihnen als »in Gebrauch« markiert sind. Dies ist eine ziemliche Verschwendung von Speicherplatz. Eine automatische Speicherplatzbereinigung wäre hier sicherlich eine gute Idee.

Speicherbereinigung wirft jedoch ein zweites Problem auf, nämlich Zombieknoten. Zombieknoten sind Knoten, die als »in Gebrauch« markiert sind, aber keine Verknüpfung mehr zum Baum haben und daher nicht mehr referenziert werden können (siehe Abbildung 2.10).

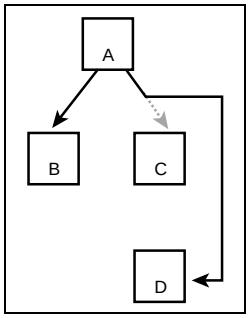

Abbildung 2.10: Zombieknoten in einer Baumstruktur

Da die Bibliothek alle Informationen über die Knoten in der Baumstruktur und ihre interne Verknüpfung enthält, lassen sich Zombieknoten sehr einfach ausmachen. Dies fehlt jedoch leider noch im Programmcode.

Das dritte Problem ähnelt dem der Zombieknoten: unterbrochene Verknüpfungen. Unterbrochene Verknüpfungen sind Verknüpfungen, die von einem Knoten ausgehen und auf einen nicht verwendeten bzw. nicht existenten Knoten zeigen (siehe Abbildung 2.11). Die Verknüpfungen werden »unterbrochen«, sobald Sie einen Knoten von dem Baum abtrennen, ihn als »frei« markieren bevor Sie alle anderen Knoten geändert haben, die auf diesen Knoten referenzieren.

Auch dies lässt sich durch eine strenge Überprüfung der Bibliotheksfunktionen und die Speicherbereinigung beseitigen.

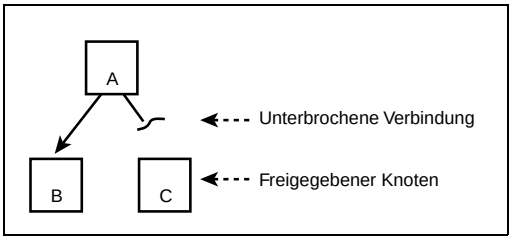

Abbildung 2.11: Unterbrochene Verknüpfungen in einer Baumstruktur

`tree_link_left()` und `tree_link_right()` verknüpfen einen linken bzw. rechten Tochterknoten, indem sie den zugehörigen Handle den Eigenschaften `$idx_left` und `$idx_right` in der Knotenstruktur zuordnen. Sie finden die Gegenparts in `tree_get_left()` und `tree_get_right()`. Diese lesen den Handle der linken bzw. rechten Verknüpfung des Knotens aus. Zusätzlich ermittelt `tree_get_parent()` den Elternknoten eines Unterknotens.

Um den Inhalt in der Baumstruktur zu speichern und Daten aus ihr auszulesen, können Sie `tree_assign_node_contents()` und `tree_retrieve_node_contents()` verwenden. Auch hier ist der dynamische Charakter von PHP hilfreich, da wir unsere Knoten nicht auf bestimmte Datentypen festlegen müssen. Es ist inzwischen in C++ allgemein üblich, Instanzen mithilfe von Klassenvorlagen zu erzeugen, die z.B. Baumklassen nur für ganze Zahlen generieren. Sie könnten zwar unzählige Baumstrukturen für eine beliebige Anzahl von Datentypen entwerfen, doch dynamisch eingegebene Inhalte können mit dieser Methode nur schwer gespeichert werden. PHP akzeptiert gemischte Typen problemlos, wodurch Sie alle Knotendatentypen jederzeit ändern können.

Übung: Fügen Sie eine zuverlässige Speicherbereinigung in Ihre Baumbibliothek ein.

Hinweis: Sie können in der Knotenstruktur einen neuen Flag für die Referenzzählung einführen. Dies erleichtert und beschleunigt die Speicherbereinigung.

Unterschätzen Sie die Übungen nicht. Zu wissen, wie etwas funktioniert, ist etwas ganz Anderes, als in der Lage zu sein, etwas zu tun. Wir empfehlen Ihnen wärmstens, zumindest zu versuchen, die Bibliothek zu verbessern. Es ist absolut keine Zeitverschwendung, selbst wenn Sie keinen Erfolg haben. Um es mit den Worten Marie Freifrau von Ebner-Eschenbach zu sagen: »Für das Können gibt es nur einen Beweis: das Tun.«

Listing 2.2 zeigt die vollständige Implementierung der Baumbibliothek:

```
//
// This structure keeps a tree node
//
class tree_node
{
    // array indices linking to neighboring nodes
    var $idx_up;
    var $idx_left;
    var $idx_right;

    var $free;

    // contents of this node, this is a mixed variable
    var $contents;
}

function tree_create()
{

    // create a new, empty array
    $return_array = array();

    // allocate the root node
    $root_node = new tree_node;

    // all other linking indices are invalid
    $root_node->idx_up = -1;
    $root_node->idx_left = -1;
    $root_node->idx_right = -1;

    // this node is unused
    $root_node->free = 1;

    // create dummy contents
    $root_node->contents = "";

    // assign root element to array
    $return_array[0] = $root_node;

    // return it back to caller
    return($return_array);

}

function tree_allocate_node(&$tree_array)
```

```
{

    // find a free node
    for($i = 0; $i < count($tree_array); $i++)
    {
        // retrieve node from array
        $node = $tree_array[$i];

        // is it in use?
        if($node->free)
        {
            // no, it is not in use, allocate it
            $node->free = 0;

            // assign node back to array to update the tree
            $tree_array[$i] = $node;

            // now return this node's index as handle
            return($i);
        }
    }

    // we haven't found a free node, so allocate a new one
    $node = new tree_node;

    // invalidate all indices
    $node->idx_up = -1;
    $node->idx_left = -1;
    $node->idx_right = -1;

    // this node is NOT free
    $node->free = 0;

    // assign dummy contents
    $node->contents = "";

    // now add this node to the tree array
    $tree_array[] = $node;

    // return new index as handle
    return(count($tree_array) - 1);

}

function tree_free_node(&$tree_array, $handle)
```

```
{

    // retrieve node from tree
    $node = $tree_array[$handle];

    // check if it is really allocated
    if($node->free)
        // this node is free, return an error code
        // note that this only serves diagnostic
        // purposes since it wouldn't hurt the tree
        // if we'd just mark it as free
        return(-1);

    $node->free = 1;

    // assign node back to tree
    $tree_array[$handle] = $node;

    return(1);

}

function tree_link_left(&$tree_array, $link_to, $child)
{

    // retrieve nodes
    $link_node = $tree_array[$link_to];
    $child_node = $tree_array[$child];

    // check if nodes are allocated
    if($link_node->free || $child_node->free)
        // return error, we do not allow linkage
        // of free nodes
        return(-1);

    // link nodes together
    $link_node->idx_left = $child;
    $child_node->idx_up = $link_to;

    // write nodes back into the array
    $tree_array[$link_to] = $link_node;
    $tree_array[$child] = $child_node;

    // return success
    return(1);
```

```
}

function tree_link_right(&$tree_array, $link_to, $child)
{

    // retrieve nodes
    $link_node = $tree_array[$link_to];
    $child_node = $tree_array[$child];

    // check if nodes are allocated
    if($link_node->free || $child_node->free)
        // return error, we do not allow linkage
        // of free nodes
        return(-1);

    // link nodes together
    $link_node->idx_right = $child;
    $child_node->idx_up = $link_to;

    // write nodes back into the array
    $tree_array[$link_to] = $link_node;
    $tree_array[$child] = $child_node;

    // return success
    return(1);

}

function tree_get_parent(&$tree_array, $handle)
{

    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // node is allocated, return its parent
    return($node->up);

}

function tree_get_left(&$tree_array, $handle)
{
```

```
    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // node is allocated, return its left child
    return($node->left);

}

function tree_get_right(&$tree_array, $handle)
{

    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // node is allocated, return its left child
    return($node->right);

}

function tree_assign_node_contents(&$tree_array, $handle, $contents)
{

    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // assign contents to node
    $node->contents = $contents;

    // assign node back into array
    $tree_array[$handle] = $node;
```

```
    // return success
    return(1);

}

function tree_retrieve_node_contents(&$tree_array, $handle)
{

    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // return contents of this node
    return($node->contents);

}
```

Listing 2.2: Implementierung der Baumbibliothek

2.6 Assoziative Arrays

Eine weitere grundlegende Struktur der Programmiersprachen stellen Arrays dar. Sie bieten die Möglichkeit, einen festen Satz (oder eine Sammlung) von Daten desselben Datentyps zu speichern, wodurch die Elemente in ihrem Datensatz über einen eindeutigen Schlüssel indexierbar werden.

In typischen »konventionellen« Programmiersprachen werden Arrays auf folgende Weise verwaltet:

```
int my_integer_array[256];          // allocate 256 integers in this array
```

Dieser Ausschnitt eines C-Programmcodes deklariert ein Arrray namens `my_integer_array`, das 256 ganze Zahlen enthält. Sie können jede dieser ganzen Zahlen adressieren, indem Sie das Array mit einer Ordnungszahl indexieren, wobei die Zahl in diesem Fall zwischen 0 und 255 liegen muss. (C beginnt die Zählung bei 0; die Zahl in der Array-Definition gibt die Anzahl der ganzen Zahlen an, die verfügbar sein sollen.) Die Indexierung erfolgt folgendermaßen:

```
int my_integer = my_integer_array[4];
```

Hierdurch wird das fünfte Element (denken Sie daran, C beginnt die Zählung bei 0) aus dem Array ausgelesen und in `my_integer` gespeichert.

Aufgrund des Wesens kompilierter Sprachen waren Sie stets an die vorausgegangene Definition Ihrer Variablen gebunden. Wenn Sie im obigen Array plötzlich mehr als 256 ganze Zahlen benötigen, hätten Sie leider Pech. Natürlich hätten Sie diese Variable als Zeiger auf ein ganzzahliges Array definieren und diesem 257 Elemente zuweisen können. Aber was wäre, wenn Sie plötzlich ein weiteres Element benötigten? Sie müssten einen neuen Speicherbereich zuweisen, den alten Inhalt des Arrays kopieren und den alten, nunmehr nicht mehr verwendeten Speicherbereich freigeben.

PHP verfolgt einen anderen Ansatz. Da PHP keine typischen Variablendeklarationen kennt (nur Typdefinitionen), werden neue Variablen quasi im Vorübergehen zugewiesen. Sobald Sie eine neue Variable generieren, indem Sie ihren Namen in den Namensraum einfügen, schaffen Sie einfach Speicherplatz, der an diesen Namen gebunden ist, nichts weiter. Die Art der Daten, die in diesem Speicherbereich abgelegt werden, ist nicht auf einen bestimmten Variablentyp beschränkt. Die Datentypen können jederzeit neu interpretiert werden, ihre Größe kann verändert werden, sie können neu zugewiesen werden und vieles mehr.

Betrachten wir uns folgendes Beispiel:

```
$my_var = 1;
$my_var = "Used to be an integer";
$my_var = array("Oh well, I like arrays better");
```

Die erste Zeile erzeugt eine neue Variable `$my_var`. PHP wird feststellen, dass ihr eine ganze Zahl zugeordnet wird und setzt daher den ursprünglichen Typ von `$my_var` auf ganze Zahl. Die zweite Zeile überschreibt den Inhalt von `$my_var` mit einer Zeichenkette. Bei einer konventionellen Programmiersprache würde dieses bei der Kompilierung zu einem Fehler führen oder zumindest zu einem Programmabbruch während der Laufzeit. PHP ändert den Typ von `$my_var` jedoch dynamisch auf »string« und ordnet die Variable neu zu, so dass genügend Speicherplatz für die Zeichenkette zur Verfügung steht. Die Zeile ändert dann den Typ von `$my_var` erneut, indem es ein Array davon erzeugt. PHP meistert alle Fälle transparent und problemlos. (Wir wissen, dass es da draußen noch weitere Sprachen ohne strikte Variablentypen gibt, aber wir möchten diese hier nicht als konventionelle Sprachen klassifizieren.)

Hinweis: PHP 3.0 hat keine vernünftige Speicherbereinigung. Bei der Neuzuweisung einer Variablen wird der bereits zugewiesene Speicherplatz nicht immer wieder verwendet. Bei langfristigen Skripten (oder Skripten, die eine umfangreiche Verarbeitung haben) könnte dies zu größeren Bereichen »toten Speichers« führen. Bei der Verwendung von speicherintensiven Skripten, die für einen langen Zeitraum genutzt werden, sollten Sie die Speicherbelegung in einer Testumgebung überwachen, bevor Sie die entsprechenden Skripte in einer produktiven Umgebung implementieren, um sicherzugehen, dass Ihr Server nicht überlastet wird. PHP 4.0 ist nicht so anfällig für dieses Problem.

Da PHP keine formalen Variablendeklarationen benötigt, ist die Verwendung der Variablen vollständig dynamisch. Ein Sonderfall der dynamischen Variablenverarbeitung in PHP sind die Arrays. Wahrscheinlich kennen Sie den herkömmlichen Array-Typ, das indexierte Array. Indexierte Arrays sind Arrays, die durch Ordnungszahlen indiziert sind. Diese Ordnungszahlen gehen in der Regel von 0 bis n, wobei n der höchstmögliche Index ist. Sprachen wie Pascal lassen eine Indexierung mit verschiedenen Bereichen, wie etwa von 3 bis 18 zu, wobei diese Bereiche später zur Laufzeit auf 0-basierte Indizes zurückgewandelt wird. Das Hauptmerkmal dieser indexierten Arrays ist, dass sie andere Indizes von einem bestimmtem Basisindex berechnen können. Nehmen wir beispielsweise an, Sie möchten bei Index 2 beginnend drei aufeinanderfolgende Array-Elemente auslesen:

```
$base_index = 2;

for($i = $base_index; $i < $base_index + 3; $i++)
    print("Element $i is $my_array[$i]<br>");
```

Bei jeder Iteration der `for()`-Anweisung berechnet dieser kleine Programmteil den nächsten Index in dem Array durch Heraufsetzen von `$i`.

Assoziative Arrays haben diese Funktion nicht. Die Besonderheit bei assoziativen Arrays besteht darin, dass Sie mit Nicht-Ordnungszahlen wie etwa Zeichenketten indiziert werden können. Jeder Zeichenkette, die als Index verwendet wird, ist ein Wert zugewiesen, was auch den Namen »assoziatives Array« erklärt. Wie Sie sich vielleicht denken können, lässt sich auch durch Zuteilen eines Basisindexes der nächste gültige Index in dem Array nicht erraten. Sie müssen die Array-Schlüssel kennen, um die zugehörigen Werte auszulesen.

Daneben können die bereits zuvor besprochenen Funktionen `list()` und `each()` verwendet werden, um assoziative Arrays zu verarbeiten.

Indizierte Arrays sind in PHP nur eine Sonderform der assoziativen Arrays. Wenn Sie `unset()` auf eines der Elemente in einem indexierten Array anwenden, bleiben alle anderen Elemente (und ihre Reihenfolge) unberührt, jedoch entsteht ein unzusammenhängendes Array. Einzelheiten hierzu finden Sie in den bereits besprochenen Abschnitten über `list()` und `each()`.

2.6.1 Mehrdimensionale Arrays

Wie der Name vermuten lässt, besitzen mehrdimensionale Arrays mehr als eine Dimension. Eindimensionale Arrays haben die folgende Form, die für Arrays wohl am bekanntesten ist:

```
$my_array[0] = 1;
$my_array[1] = 777;
$my_array[2] = 45;
```

Um diesem Array-Typ zu indizieren, brauchen Sie nur einen Index, der die Anzahl der möglichen Werte auf den Bereich dieses Index begrenzt. Häufig ist es jedoch sinnvoll, mehrdimensionale Arrays zu erzeugen, wenn Sie komplexe Datensätze verwalten. Typische Beispiele hierfür sind Bitmaps und Bildschirmpuffer. Wenn Sie auf Ihren Bildschirm blicken, sehen Sie (zumindest heutzutage) eine zweidimensionale Projektion Ihres Desktops. Die Fenster, Bitmaps, Befehlszeilen, Cursor, Zeiger, alles ist zweidimensional. Um diese Daten auf bequeme Weise darzustellen, könnten Sie natürlich alles in Arrays mit einer einzigen Dimension zusammenfassen. Es ist jedoch passender, Arrays zu verwenden, welche dieselbe Anzahl von Dimensionen haben wie die Eingabedaten. Um beispielsweise eine Bitmap (eine Reihe von Pixel) für einen Mauszeiger zu speichern, könnten Sie in Ihrem Array einen weiteren Index hinzufügen:

```
// clear mouse bitmap
for($x = 0; $x < MOUSE_X_SIZE; $x++)
    for($y = 0; $y < MOUSE_Y_SIZE; $y++)
        $mouse_bitmap[$x][$y] = 0;
```

Hiermit wird eine Maus-Bitmap gelöscht und alle Elemente auf 0 gesetzt. Dabei werden zwei Schleifen verwendet, eine für jede Dimension. Abbildung 2.12 zeigt eine grafische Darstellung von zweidimensionalen Daten mit Daten aus einem Koordinatensystem. Intern, d.h. im Speicher, werden die Datenelemente natürlich in Serie gespeichert (der RAM hat nur eine Dimension für die Indizierung). Eine passende Analogie für die Visualisierung bietet das Koordinatensystem.

In Arrays gibt es keine Begrenzung für die maximale Anzahl der Dimensionen (offen gesagt haben wir es noch nicht ausprobiert, aber es wird kaum eine Anwendung für Arrays mit 16 Dimensionen oder mehr geben). Dimensionen

können auch in verschiedenen Typen auftreten (die erste Dimension assoziativ, die zweite Dimensionen mit ganzzahligen Indizes, die dritte wieder assoziativ usw.). Sie eignen sich damit auch hervorragend für die Darstellung statistischer Daten.

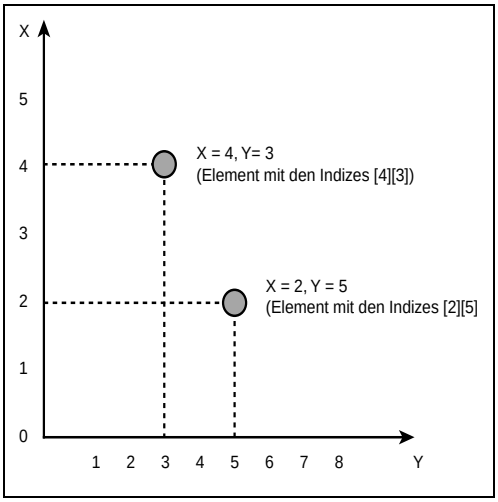

Abbildung 2.12: Eine zweidimensionale Array-Struktur

2.6.2 Variable Argumente

Bei der Verwendung von Funktionen ist es häufig notwendig, mehr als einen Wert zurückzugeben oder die gegebenen Parameter zu ändern. `fsockopen()` gibt beispielsweise als Rückgabewert den Sockel-Bezeichner zurück, denkbar wäre jedoch auch die Ausgabe eines Fehlercodes mit beschreibendem Text, z. B. bei potentiellen Fehlern:

```
// try to open a socket for HTTP with a 30 second timeout
$socket_handle = fsockopen("www.myhost.com", 80, $error_nr, $error_txt,
30);
if(!$socket_handle)
{
    print("Couldn't connect to HTTP host.<br>");
    print("Error code: $error_nr, Reason: $error_txt<br>");
}
```

Wenn eine Verbindung zum gewünschten Host nicht errichtet werden konnte, gibt dieser Code einen Fehlercode und den Fehlergrund aus. Die Variablen wurden ursprünglich als Parameter an `fsockopen()` übergeben. Da diese Parameter in der `fsockopen()`-Deklaration jedoch als »durch Referenz weitergegeben« deklariert sind, kann `fsockopen()` sie modifizieren und die Änderungen anschließend global zur Verfügung stellen.

In der Regel greifen Funktionen auf ihre Parameter nicht durch Referenz zu. Wenn sie ihre Werte während der Programmausführung ändern, arbeiten sie mit einer lokalen Kopie des ursprünglichen Wertes.

```
function calculate($a, $b, $c)
{

    $a = $b + $c;

}

$i = 1;
$j = 2;
$k = 3;

print("I $i, J $j, K $k<br>");
calculate($i, $j, $k);
print("I $i, J $j, K $k<br>");
```

Beide Druckanweisungen geben denselben Inhalt aus. Während `calculate()` bei der Programmausführung $a modifiziert, ändert sich der Inhalt von $i nicht, obwohl er als Argument für den Parameter $a übergeben wurde. Dies liegt daran, dass `calculate()` nicht mit der Variablen selbst, sondern mit einer Kopie der ursprünglichen Variablen arbeitet. Sobald die Funktion beendet ist, wird die Kopie der Variablen verworfen, mit welcher die Anweisung gearbeitet hat, und der Inhalt geht verloren.

Wie im Falle von `fsockopen()` kann es manchmal wünschenswert sein, die an einem Parameter vorgenommene Änderung zu speichern und sie global zur Verfügung zu stellen. Damit dies möglich wird, muss die Variable jedoch nicht als Kopie sondern als Referenz übergeben werden. Damit erhält die Funktion einen Zeiger auf den Speicherblock, in dem die ursprüngliche Variable abgelegt ist. Mithilfe dieses Zeigers kann die Funktion auf die globale Instanz der Variablen zugreifen und diese direkt ändern:

```
function calculate(&$a, $b, $c)
{

    $a = $b + $c;

}

$i = 1;
$j = 2;
$k = 3;

print("I $i, J $j, K $k<br>");
calculate($i, $j, $k);
print("I $i, J $j, K $k<br>");
```

Wie Sie hier sehen, ist hier nur ein Zeichen anders: das Ampersand (`&`), das im vorherigen Programmcode fehlte. Dieses, einem Funktionsparameter vorangestellte Zeichen gibt an, dass er per Referenz übergeben wurde.

Hinweis: Um ein Ampersand (`&`) einzufügen, müssen Sie nicht die Zeile ändern, in welcher die Funktion aufgerufen wird. PHP wandelt Ihre Parameter automatisch in Referenzen um, wenn es eine Funktion findet, die Parameter auf diese Weise übergeben möchte.

Bei der Änderung von $a durch `calculate()` wird die lokale Kopie von $i, demnach nicht verändert, statt dessen wird durch den Zugriff auf den globale Speicher von $i die Änderung direkt durchgeführt. Viel wichtiger ist aber, dass die Änderungen, die an $i vorgenommen wurden, bei Beendigung der Funktion nicht verloren gehen, was beim Arbeiten mit einem lokalen Speicher nicht funktioniert.

Die Parameterübergabe durch Referenz ist häufig eine nützliche Methode, wenn Sie mit einer Funktion mehr als einen Wert ausgeben möchten oder Ihre Variablen durch mehrfachen Aufruf einer Funktion auf einmal ändern wollen, um eine komplexe Berechnung in einem Durchgang durchzuführen. Sie sollten es jedoch vermeiden, Strukturen aufzusplitten und in Parameterlisten zu zwängen, wenn Sie die Daten genauso gut direkt in einer Struktur zurückgeben könnten. Überstrapazieren Sie diese Art von Parameter jedoch nicht. Diese Maßnahme sollte wirklich eine Ausnahme bleiben, denn sie ist keine ganz saubere Programmierpraxis (Funktionen ändern ihre Parameter normalerweise nicht auf globaler Ebene, und dadurch schleichen sich manchmal hässliche Fehler ein), auch wenn es die Dinge häufig erleichtert und Sie damit clevere Tricks anwenden können.

Ein mögliches Beispiel ist die »automatische« Aktualisierung sogenannter »Laufvariablen«. Laufvariablen sind Variablen, die ihre Werte während einer algorithmischen Schleife ändern. (Ein Sonderfall hiervon sind Zählvariablen in `for()`-Anweisungen.) Ein konkretes Beispiel, das hiervon profitiert, ist der Lauflängen-Kodierungsalgorithmus (RLE) , der weithin bekannt ist, da er in Formaten wie ZSoft's PCX Bildformat verwendet wurde (und noch verwendet wird).

Der RLE-Algorithmus ist ein Komprimierungsalgorithmus, der von der Tatsache profitiert, dass viele Farbbilder mit niedriger Auflösung dieselben Datenbytes wiederholt abspeichern (insbesondere Bitmaps, die nur zwei Farben – schwarz und weiß – enthalten). Sehen wir uns einmal diese Darstellung eines einfachen Rechtecks an:

```
11111111111111111111
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
11111111111111111111
```

Mit ein wenig Vorstellungskraft sehen wir ein Rechteck aus Nullen mit einem Rand aus Einsen. Wenn Sie diesen in eine Datei wie »vorliegend« übertragen wollten, benötigten Sie 20 x 10 = 200 Elemente (20 Spalten mal 10 Zeilen). Wir gehen an dieser Stelle einmal davon aus, dass ein Element mindestens einem Byte entspricht und lassen außer acht, dass eine einfache Komprimierung bereits durch Packen dieser Daten in Bits erreicht werden kann.

Was sich hier zeigt, ist, dass die Speicherung der Daten wie vorliegend nicht unbedingt sinnvoll ist. Wenn Sie die Daten jemandem zum Niederschreiben diktieren sollten, würden Sie es nicht als »Eins Eins Eins Eins Eins Eins Eins [...] Eins Null Null Null Null Null [...] Null Eins Eins Eins [...],« vorlesen, sondern würden dem Schreiber »zwanzig Einsen, eine weitere Eins, achtzehn Nullen, [...]« diktieren.

Das Gleiche macht auch der RLE-Algorithmus, der die Anzahl der aufeinanderfolgenden Elemente gleichen Wertes zählt und sie anschließend als »Zähler/Wert«-Paar abspeichert. Nach der Komprimierung könnten die obigen Daten dann folgendermaßen aussehen:

```
21, 1, 18, 0, 2, 1, 18, 0, [...], 18, 0, 21, 1
```

Das erste Element gibt stets den Zähler an, während das zweite Element das Datenelement darstellt. Um dies wieder zu dekomprimieren, müssen Sie lediglich den Zähler lesen und das folgende Datenelement so häufig ausgeben, wie es der Zähler angibt. Mit diesem »Trick« reduzieren Sie die Anzahl der benötigten Elemente von 200 auf 34.

Das Problem bei diesem Algorithmus ist, dass er bei vielen verschiedenen Elementen in einer Zeile mehr Ausgabedaten erzeugt, als ursprünglich Eingabedaten vorhanden waren. Durch Speichern vieler Elemente, die einen Datenzähler von Eins haben, wird dieser Algorithmus damit schnell ineffektiv.

Diese Schwäche lässt sich mit einem kleinen Trick, der sogenannten »Zählerschwelle« umgehen. Jeder Datentyp hat einen spezifischen Zählerbereich: Im obigen Fall überschritt der Zähler beispielsweise niemals 21. Genaugenommen waren die häufigsten Werte 18 und 2. Wenn wir den Zähler nun auf einen Bereich von 1 bis 31 beschränken, können wir auch »reine« Zahlen einfügen, die keine Komprimierung im Eingabestrom benötigen. Der Kompres-

sor spuckt alle Eingabedatenelemente größer als 31 einfach aus, ohne ihnen einen Zähler voranzustellen. Damit erreichen wir für alle Eingabewerte größer als 31 einen 100% optimalen Algorithmus. Alle Werte bis 31 werden zwar immer noch nicht optimal komprimiert, aber dies ist vernachlässigbar.

Um die Daten zu dekodieren, ist es nun wichtig, zwischen reinen Datenelementen und komprimierten Datenelementen zu unterscheiden, was Sie auch in den Listings 2.3, 2.4 und 2.5 sehen können. Hier kommen variable Argumente optimal zum Einsatz.

```
function encode_data()
{

    // do initial setup on our variables
    $current_count = 0;
    $status = read_from_input($current_byte);
    $old_byte = $current_byte;
    $output = array();

    // as long as there's input data, loop
    while($status)
    {
     // check if the current byte matches the last one
     if($old_byte == $current_byte)
     {
         // there's a match, increase counter
         $current_count++;

         // does the counter exceed the threshold?
         if($current_count == COUNTER_THRESHOLD)
         {
          // it does, flush cache and restart
          $output[] = chr($current_count);
          $output[] = $current_byte;

          $current_counter = 0;
         }
     }
     else
     {
             // bytes don't match

             // do we have a cached pair?
             if($current_count > 1)
             {
              // yes we do, write it
              $output[] = chr($current_count);
```

```
                $output[] = $old_byte;

                $current_count = 1;
               }
               else
               {
                // we don't have a cached pair,
                // write literal
                if(ord($old_byte) < COUNTER_THRESHOLD)
               {
                    // this byte could be mistaken as a counter
                    // value, so write a dummy pair
                    $output[] = chr(1);
                    $output[] = $old_byte;
               }
               else
               {
                   // can't be mistaken as counter value, just
                   // write the value directly
                   $output[] = $old_byte;
               }
              }
        }

        // set current byte as old byte
        $old_byte = $current_byte;

        // get new byte and loop
        $status = read_from_input($current_byte);
    }

    return($output);

}
```

Listing 2.3: Der RLE-Komprimierungsalgorithmus

```
function get_encoded_pair(&$count, &$value)
{

    // check input stream
    if(!read_from_input($data_element))
    {
        // no input data available, return
        // zero count and dummy data
        $count = 0;
```

```
        // indicate failure
        return(0);
    }

    // test if this is literal data
    if(ord($data_element) >= COUNTER_THRESHOLD)
    {
        // this is literal data, return
        // count of one and data element
        $count = 1;
        $value = $data_element;
    }
    else
    {
        // this has been a count, assign it
        $count = ord($data_element);

        // try to retrieve the data element
        // itself
        if(!read_from_input($value))
        {
            // input data is corrupted,
            // return zero count
            $count = 0;

            // indicate failure
            return(0);
        }

    }

    // return success
    return(1);

}

function decode_data()
{

    // initialize output array
    $output = array();

    // decompress all data into the array
    while(get_encoded_pair(&$count, &$value))
    {
        for($i = 0; $i < $count; $i++)
```

```
            $output[] = $value;
    }

    return($output);

}
```

Listing 2.4: Der RLE-Expandierungsalgorithmus

```
//
// this declaration must exist and be equal
// both for the compressor and decompressor
//
define("COUNTER_THRESHOLD", 32);

//
// this tool function is needed for both the
// compressor and decompressor to read data
//
function read_from_input(&$data_element)
{
    // This is a dummy function to retrieve a data element
    // from the input data. It could contain code to read
    // from an array, from the standard input, or something
    // completely different.
    // As an example, this function reads from a global
    // file. (Not a good idea to have global files but
    // just for the example's sake.)
    global $file_handle;

    // check if we have reached the end of the file
    if(feof($file_handle))
    {
        // we did, return error
        return(0);
    }

    // we did not encounter the end of the file,
    // so read next element
    $data_element = fgetc($file_handle);

    // return success
    return(1);
}

// include compressor and decompressor
include("compressor.php3");
```

```
include("decompressor.php3");

// define filenames
$original_file = "data.original";
$compressed_file = "data.compressed";
$decompressed_file = "data.decompressed";

// -> all procedures need the global variable $file_handle
// (bad practice but best for a simple example)

// open input file
$file_handle = fopen($original_file, "r");
if(!$file_handle)
    die("Error opening file.");

// encode it
$output = encode_data();

// close input file
fclose($file_handle);

// open output file
$file_handle = fopen($compressed_file, "w");
if(!$file_handle)
    die("Error creating file.");

// write decoded data
for($i = 0; $i < count($output); $i++)
    fputs($file_handle, $output[$i]);

// close output file
fclose($file_handle);

// open input file
$file_handle = fopen($compressed_file, "r");
if(!$file_handle)
    die("Error opening file.");

// decode it
$output = decode_data();

// close input file
fclose($file_handle);

// open output file
$file_handle = fopen($decompressed_file, "w");
```

```
if(!$file_handle)
    die("Error creating file.");

// write decoded data
for($i = 0; $i < count($output); $i++)
    fputs($file_handle, $output[$i]);

// close file
fclose($file_handle);
```

Listing 2.5: Beispiel für die Verwendung des RLE-Prozessors

Dieses Beispiel zeigt sehr schön, wie der Dekodierer und die Leserlogik in kleinere separate Funktionen unterteilt werden können. Der eigentliche Dekodierer ist nur noch wenige Zeilen lang. Die Funktion, welche die Eingabedaten liefert, kann vom Rest des Programmcodes abgetrennt werden, und auch der »Entscheidungsalgorithmus«, der zwischen reinen und kodierten Daten unterscheidet, ist eine separate kleine und leicht zu verstehende Funktion.

Eine clevere Methode ist, reine Daten mit einem Zähler von 1 zurückzugeben, so dass sich die Dekomprimierungsschleife nicht mehr mit ihnen befassen muss. Sie kann die gelieferten Daten einfach unverändert in das Ausgabe-Array schreiben.

Die Fehlerkontrolle könnte zwar noch etwas verbessert werden, aber dies überlassen wir dem Leser. Es ist nicht sonderlich schwer.

Variable Argumentenlisten

Variable Argumentenlisten, die häufig auch als optionale Parameter bezeichnet werden, geben Ihnen die Möglichkeit, Funktionsparameter mit einem Standardwert vorzubelegen. Wenn der Anrufer keinen Wert für das Argument angibt, wird das Standardargument angenommen. Dem Anrufer kann eine Liste mit optionalen Parametern zur Verfügung gestellt werden, die er verwenden kann, aber nicht muss.

Optionale Parameter sind wie folgt definiert:

```
function open_http_connection($hostname, $port = 80, $timeout = 10)
{

    $socket = fsockopen($hostname, $port, $timeout);
    /* rest of the code goes here */

    return($socket);

}
```

```
$regular_socket = open_http_connection("www.myhost.com");
$slow_socket = open_http_connection("www.myhost.com", 80, 20);
$test_socket = open_http_connection("testserver.myhost.com", 8080);
$slow_test_socket = open_http_connection("testserver.myhost.com", 8080,
➥20);
```

Die Funktion `open_http_connection()` akzeptiert ein reguläres Argument namens `$hostname`, das den Namen des Host-Rechners angibt, mit welchem eine Datenverbindung hergestellt werden soll. Zusätzlich hat die Funktion zwei optionale Argumente `$port` und `$timeout`, welche die zu verbindende Portnummer und das Timeout für die Verbindung (in Sekunden) angeben.

Beide haben vorgegebene Standardwerte, die durch das Gleichheitszeichen (=) angegeben werden, gefolgt von dem gewünschten Standardwert. Wenn diese Argumente nicht vom Anrufer ausgefüllt werden, ersetzt PHP die fehlenden Einträge durch die Standardwerte.

Wie Sie an diesen beiden Beispielen sehen, wird im ersten Aufruf nur `$hostname` verwendet. `$port` und `$timeout` fehlen. PHP füllt diese Lücken mit den Standardwerten aus, so dass der Aufruf nun folgendermaßen aussieht:

```
$regular_socket = open_http_connection("www.myhost.com", 80, 10);
```

Sie können trotzdem ihrerseits die optionalen Parameter angeben und die Standardwerte überschreiben, wie der zweite Aufruf zeigt. Der Wert für `$port` ist immer noch 80, aber `$timeout` ist nun auf 20 Sekunden gesetzt.

Sie müssen die Standardargumente angegeben, deren Wert Sie ändern möchten, wie das dritte Beispiel zeigt. Hier ist nur `$port` als 8080 angegeben. `$timeout` fehlt und bleibt daher auf seinem Standardwert von 10 Sekunden.

Die Tatsache, dass PHP keine Möglichkeit hat herauszufinden, welcher Wert zu welchem Parameter gehört, bedeutet, dass Sie alle optionalen Parameter am Ende der Argumentenlisten anfügen müssen. Wenn Sie `$hostname` als einzigen obligatorischen Parameter am Ende der Liste einfügen würden (und `$port` und `$timeout` davor beließen), würde PHP beim ersten Aufruf, bei dem nur `$hostname` angeben ist, vermuten, dass die Zeichenkette für den Namen des Hostrechners der Wert für `$port` wäre, was zu erheblicher Verwirrung führen würde.

Genauso wenig können Sie einfach willkürlich optionale Parameter auswählen, für die Sie Werte liefern möchten. Wenn Sie eine Funktion haben, die drei optionale Parameter hat, und Sie nur den letzten Parameter in der Argumentenliste ändern wollen, müssen Sie trotzdem die Standardwerte für die ersten beiden Parameter angeben. (Dies können Sie auch im zweiten der obigen Beispiele sehen, in dem nur `$timeout` geändert wird.)

PHP 4.0 kann Argumente wirklich variabel verwalten. Eine Funktion kann mehr Argumente haben als die Funktionsdefinitionslisten, und Sie können mit den Funktionen `func_get_args()`, `func_num_args()` und `func_get_arg()` auf eine beliebige Anzahl von Argumenten zugreifen.

Der Rückgabewert von `func_get_args()` ist ein indiziertes Array, das von links nach rechts mit allen Argumentwerten aufgefüllt wird, die an die Funktion übergeben wurden:

```
function show_arguments()
{

    $argument_array = func_get_args();

    for($i=0; $i<count($argument_array); $i++)
    {
        print("$i => $argument_array[$i]<br>");
    }

}

show_arguments("Leftmost", "Middle", "Rightmost");
```

Die Funktion `func_num_args()` gibt die Anzahl der übergebenen Argumente zurück. `func_get_arg()` gibt ein spezifisches Argument zurück. `func_get_arg(0)` würde beispielsweise das erste Argument zurückgeben.

Variable Variablennamen

Variable Variablennamen. Für jene, die dieses Konzept noch nicht kannten (und für uns, als wir es zum ersten Mal hörten) – einfach verrückt! Variable Variablennamen dienen dazu, auf Variablen zuzugreifen, deren Namen Sie vorher noch nicht kennen und die Sie erst während der Laufzeit erstellen. Diese Funktion ist durch den Interpretationscharakter von PHP möglich. PHP wandert einfach den Programmcode ab und übersetzt alles, was es findet, in etwas möglichst Sinnvolles. Der folgende Programmcode zeigt ein einfaches Beispiel für variable Variablennamen:

```
<?

$my_var = "hello";

$$my_var = 1;

?>
```

In der zweiten Zeile wird `$my_var` das Präfix `$$` vorangestellt. Dies ist das Grundprinzip von variablen Variablennamen. Natürlich könnten Sie variable Variablennamen verschachteln, um variable variable Variablennamen zu erhalten usw. Aber auch mit variablen Variablennamen können Sie einige nette Sachen anstellen.

Ein Beispiel aus dem Leben: phpPolls, das Stimmabgabeprogramm, das wir in Kapitel 1 »Entwicklungskonzepte« beschrieben haben, verwendet variable Variablennamen. Um zu verhindern, dass Wähler in derselben Wahl mehrfach abstimmen, basiert einer der Schutzmechanismen auf Cookies. Sobald ein Wähler wählt, wird ein Cookie gesetzt und mit einem Namen versehen, der einen konfigurierbares Präfix und eine eindeutige ID erhält, welche die Wahl kennzeichnet. Da Cookies wieder in den globalen Namensraum zurückgeführt werden, prüft phpPolls bei der Stimmabgabe eines Benutzers, ob es eine globale Variable mit dem Namen des Cookies gibt, den es vorher erstellt hatte. Wenn dies der Fall ist, nimmt phpPolls die Stimmabgabe nicht an.

Für diese Aufgabe sind variable Variablennamen sehr gut geeignet. Hier ein Auszug aus dem phpPolls-Quellcode:

```
$poll_object = mysql_fetch_object($poll_result);
$poll_timeStamp = $poll_object->timeStamp;

$poll_cookieName = $poll_cookiePrefix.$poll_timeStamp;

// check if cookie exists
if(isset($$poll_cookieName))
{
    // cookie exists, invalidate this vote
    $poll_voteValid = 0;
}
else
{
    // cookie does not exist yet, set one now
    setCookie("$poll_cookieName", "1", $poll_cookieExpiration);
}
```

Zunächst wird die eindeutige ID für den Cookie gelesen, die aus dem Zeitstempel für die Wahl besteht, und dann mit dem Präfix des Cookies `$poll_cookiePrefix` zu dem Namen der gewünschten Variablen `$poll_cookieName` zusammengesetzt. Mithilfe von `isset()` wird das Vorhandensein der Variable (und damit des Cookies) überprüft und eine entsprechende Aktion eingeleitet.

Variable Funktionsnamen

Was wir über variable Variablennamen gesagt haben, gilt auch für Funktionsnamen. Auch Funktionsnamen können mithilfe von Variablen erzeugt werden, was eine dynamische Datenverarbeitung, Installation von modifizierbaren Rückrufen u.ä. ermöglicht. Anstatt Funktionsnamen fest im Programm einzugeben, können Sie Zeichenkettenvariablen verwenden, um die Funktion anzugeben, die Sie aufrufen möchten:

```
function my_func($a, $b)
{

    print("$a, $b");

}

$function = "my_func";

$function(1, 2);
```

Nach der Deklaration von `my_func()` wird die Variable `$function` der Zeichenkette `my_func` zugeordnet. Da diese Zeichenkette mit dem Namen der Funktion identisch ist, die Sie aufrufen möchten, können Sie diese beim Aufruf von `my_func()` verwenden.

Dies ist natürlich ein recht einfaches Beispiel für variable Funktionsnamen. Sie sind sehr nützlich, wenn Sie, je nach Anzahl der variablen Flags, zwischen mehreren Funktionen hin- und herschalten müssen.

Angenommen, Sie wollten Email-Anhänge dekodieren. Diese können verschieden Formate haben, z.B. base64 oder uuencoded, um nur zwei Beispiele zu nennen. Wenn Sie einen »geschlossenen« Parser generieren, der nur eine der beiden Kodierungen erkennt, lässt sich dieser nur schwer erweitern. Sobald Sie neue Formate benötigen, stecken Sie fest. Dies ist ein klassischer Fall für variable Funktionsnamen:

```
function decode_base64($encoded_data)
{

    // do something with the encoded data

    return($decoded_data);

}

function decode_uuencoded($encoded_data)
{
```

```
    // do something with the encoded data

    return($decoded_data);

}

$mail_text = fetch_mail();
$encoder_type = determine_encoding($mail_text); // returns: "base64" for
                                                    ➥Base64
                                                // returns: "uuencoded" for
                                                    ➥UUEncoded
$decoder = "decode_".$encoder_type;

$decoded_data = $decoder($mail_text);
```

Dieser Programmcode stellt automatisch die korrekte Bearbeitungsroutine für die Eingabedaten fest. `determine_encoding()` gibt die Zeichenfolge zurück, die den zu entschlüsselnden Datentyp angibt, für den eine entsprechende Funktion vorhanden sein muss. Der Name der aufzurufenden Funktion wird dann in `$decoder` geschrieben und sofort aufgerufen.

Der Nachteil dieses Verfahrens ist, dass es nicht sehr sauber ist. Sie werden kaum ein »Standard« verhalten feststellen. Das Entschlüsselungsverfahren ist vollständig dynamisch und könnte zusammenbrechen, wenn `determine_encoding()` ein bedeutungsloses Ergebnis erzeugt. Es ist jedoch eine recht komfortable Art und Weise Eingabedaten zu verarbeiten. Sobald neue Kodierungstypen erscheinen, brauchen Sie nur eine Funktion mit passendem Namen zu erzeugen und `determine_encoding()` anzupassen, um die entsprechende Zeichenfolge zu erhalten.

Solange Sie `determine_encoding()` absturzsicher gestalten, so dass es eine sinnvolle Zeichenfolge zurückgibt (selbst wenn es nur ein Dummy ist), sind wir mit dieser Methode vollkommen einverstanden. Solange Sie sicherstellen können, dass Ihr Skript sich während der gesamten Laufzeit in einem definierten Status befindet, ist nichts dagegen einzuwenden, diese dynamische Datenverwaltung für Produktionsumgebungen einzusetzen.

Ein realistisches Beispiel eines Skripts, das extensiven Gebrauch von variablen Funktionsnamen macht, ist phpIRC (das wir im folgenden Kapitel besprechen werden). phpIRC ist eine IRC-Schicht für PHP, die über eine benutzerfreundliche API einen Zugriff auf IRC(Internet Relay Chat)-Netzwerke bietet. Da die Verwaltung der Eingabedaten nicht-linear ist und vollständig vom Benutzer abhängt, verwendet phpIRC für jedes ankommende Paket eine Reihe von Ereignisklassifizierungen. Der Benutzer kann für jedes Ereignis in phpIRC Bearbeitungsroutinen installieren, die auf jede Art von ankommenden Daten angemessen reagieren. phpIRC speichert die Namen der Rückruffunktionen in einem internen Array. Sobald Daten ankommen, durchsucht

es sein Rückruf-Array nach Funktionen, die auf diesen entdeckten Datentyp passen und ruft alle passenden Funktionen auf einmal auf. Dies ermöglicht, ähnlich wie bei dem vorangegangenen Email-Beispiel, eine dynamische Datenverarbeitung, was sich bezahlt macht, wenn Sie nur zur Laufzeit entscheiden können oder wollen, wie Sie mit ankommenden Ereignissen verfahren.

Wenn Sie diesen Ansatz weiter verfolgen, können Sie variable Funktionsnamen einsetzen, um das Verhalten Ihrer Skripte zur Laufzeit zu ändern und gleichzeitig benutzerdefinierte Plugins installieren, welche zur Laufzeit automatisch an den Programmcode angehängt werden Damit erhalten Sie zusätzliche Funktionen für das Skript, ohne dass Sie dafür eine einzige Programmzeile ändern müssen.

2.7 Polymorphisierung und selbstmodifizierender Code

Die Nachteile von variablen Funktionsnamen (und teilweise von variablen Variablennamen) ist, dass Sie stets einen »festen« Programmteil haben müssen (im Fall der variablen Funktionsnamen eine Liste mit zuvor deklarierten Funktionen, die Sie verwenden können) und einen »variablen« Teil (der Teil, der die Funktionsnamen in eine Variable umwandelt und dann die Funktion aufruft, dessen Namen erstellt wurde). Dies bedeutet, dass Sie für jeden möglichen erstellbaren Namen vorher eine Funktion generieren müssen, damit das Programm korrekt funktioniert – was eine Form von Beschränkung darstellt.

Dies können Sie durch vollständig dynamische Programme umgehen, Programme die sich selbst »en passant« generieren. Diese Idee stammt ursprünglich aus den »frühen« Tagen der Programmierung und wurde z. T. von Spieleprogrammierern und Virusschreibern erfunden.

Es begann alles mit sich selbst modifizierendem Programmcode. Bei den inneren Schleifen von Spielen, beispielsweise den Prozeduren, die zum Kopieren des Pufferinhalts in den Bildschirmspeicher dienten, war und ist Zeit der kritische Faktor. Da die Verarbeitungszeit jedoch keine unerschöpfliche Quelle war und ist, mussten sich die Entwickler etwas einfallen lassen, um das Beste aus ihrem Werk herauszuholen.

Häufig mussten in der innersten Schleife eine Reihe von Entscheidungen getroffen werden. Wenn beispielsweise die Kopierroutine des Puffers bei bestimmten Ereignissen nur jede zweite Zeile in den Bildschirmspeicher ausgeben sollte, mussten Sie ein paar `if()/then`-Konstrukte in einen Programmteil einbetten, der diese am wenigsten benötigte. Diese Konstrukte kosteten

wertvolle Verarbeitungszeit, und da diese innerste Routine etwa 80% der gesamten Verarbeitungszeit ausmachte, bedeute eine Halbierung dieser Verarbeitungszeit durch Entfernen dieser Konstrukt eine 40%ige Steigerung der verfügbaren Rechenkapazität.

Für jeden Fall eine Reihe von Routinen zu schreiben, wäre auch nicht sehr hilfreich gewesen. Sie hätten trotzdem Speicherplatz verschwendet und die Entscheidungen nur an eine andere Stelle verschoben. Sie können damit zwar die Gesamtleistung verbessern, aber eine optimale Leistung erreichen Sie nicht.

Deshalb wurde die Technik des selbstmodifizierenden Programmcodes eingeführt. Wo ein Programmteil früher eine der Bedingungen der innersten `if()/then`-Schleife geändert hätte, indem er die zuständigen Flags entsprechen gesetzt hätte, programmierte er die innere Schleife einfach neu, und zwar so, dass sie wie gewünscht agiert, d.h. so als hätte sie die Flags ausgewertet. Die Modifikationen, die hierfür notwendig waren, betrafen häufig nur das Ändern von ein oder zwei Bytes und waren meist nicht aufwendiger als das Setzen von Flags. Dies funktionierte natürlich nur auf Maschinencode-Ebene und war extrem systemabhängig – dafür aber auch sehr mächtig.

Die Entwickler von Virusprogrammen haben dieses Verfahren schließlich auf die Spitze getrieben, indem sie polymorphe Programme entwickelten. Polymorphe Programme heißen polymorph, weil sie ihren eigenen Programmcode ändern, während sie noch dieselbe Aufgabe ausführen. Die einfachste Methode, um einen polymorphen Programmcode zu entwickeln, bestand darin, die Viren zu komprimieren und jedes Mal einen anderen Komprimierungsalgorithmus oder andere Komprimierungsparameter auszuwählen. Nach jeder Komprimierung wurde so der Bytecode geändert, doch nach der Dekomprimierung wurde das ursprüngliche Programm wiederhergestellt. (Versuchen Sie einmal die ZIP-Archive anzusehen, die dieselben Daten auf verschiedenen Komprimierungsebenen enthalten: Sie sehen ganz verschieden aus, haben aber alle nach der Expansion wieder dieselben Daten.) Der kompliziertere Ansatz war, Anweisungsblöcke dynamisch neu zu gruppieren und gleichzeitig die algorithmische Struktur beizubehalten. Dieses Verfahren erforderte manchmal sehr anspruchsvollen Programmcode, war aber auch sehr effektiv. Da jede Methode dazu führte, dass der Bytecode geändert wurde, wurden verschiedene Signaturen für die einzelnen Infektionen desselben Virus erzeugt, so dass es für Antivirus-Programme nahezu unmöglich war, sie zu entdecken, während die Viren andere Programme lustig weiter infizieren konnten.

Was hat dies mit PHP zu tun? Natürlich können Sie keine solchen polymorphen Programme erstellen. Die PHP-Architektur verhindert während der Laufzeit Änderungen des Programmcodes, der bereits analysiert ist. Trotzdem enthalten sie nützliche Hinweise. Eine Möglichkeit sind dynamische Funktionsparser, auf die wir im nächsten Abschnitt eingehen.

2.7.1 Dynamischer Funktionsgenerator

Als wir an diesem Buch schrieben, fragte jemand aus der deutschen PHP-Mailingliste, ob wir eine Möglichkeit wüssten, vom Benutzer eingegebene mathematische Funktionen zu verwalten. Er wollte wissen, wie er eine vom Benutzer eingegebene Funktion mithilfe einer Webformel in PHP grafisch darstellen kann. Er wusste nicht, wie er mit dem Text verfahren sollte. Wie lässt sich etwas wie *f(x) = m * x + b* in ein Diagramm übertragen?

Die Diskussion führte schnell in eine traditionelle Richtung. Jeder fing an, im großen, im wirklich großen Rahmen zu denken, anstatt die einfachen, offensichtlichen Dinge in Betracht zu ziehen. Uni sono (aus dem Lateinischen: mit einer Stimme) lautete der allgemeine Vorschlag zur Lösung des Problem folgendermaßen:

1. Analysiere die Eingabedaten.
2. Generiere eine syntaktisch analysierte Darstellung (etwas in Bezug auf Compilertechniken).
3. Lasse einen Prozessor über die analysierte Darstellung laufen und erzeuge Schritt für Schritt numerische Ausgabedaten.

Für unser konkretes Beispiel *f(x) = m * x + b* sähe dies folgendermaßen aus:

1. Sie stellen fest, dass es sich um eine Funktion von *x* handelt, in der *m* und *b* Variablen sind.
2. Sie erzeugen zur einfacheren Handhabung eine Struktur, um den Text der Funktion intern darzustellen, z.B. in einer Baumstruktur, welche die Abhängigkeiten der Variablen vom Multiplikationszeichen (*) bzw. Additionszeichen (+) angibt.
3. Sie setzen für *x* (der Variablen, von der die Funktion abhängt) verschiedene Werte ein, ermitteln *y*, speichern das Ergebnis und interpolieren die Daten.

Dies ist der in den Universitäten gelehrte Ansatz, den wir bereits in komplexen Quellcodes u.ä. gesehen haben. Niemand scheint sich von gewohnten Lösungsansätzen frei machen und eine innovativere Lösung suchen zu können. Haben Sie schon einmal darüber nachgedacht, was PHP tut, wenn es ihre Skripte interpretiert?

1. Es analysiert die eingegebenen Quelldaten.
2. Es erzeugt eine analysierte Darstellung.
3. Es lässt einen Prozessor über die analysierte Darstellung laufen.

Gut, die ist etwas vereinfacht, aber im Grunde genommen ist es das, was wir brauchen. Warum sollten wir also die Eingabefunktion nicht in einen gültigen PHP-Code umwandeln und dann PHP die Arbeit für uns machen lassen? Wie Sie bereits in diesem Kapitel gesehen haben, unterstützt PHP dynamisches Programmieren, so dass sich die gesamte Aufgabe möglicherweise recht einfach bewältigen lässt.

Tatsächlich sind die regulären Ausdrücke, die zur Umwandlung einer einfachen mathematischen Funktion in PHP-Code benötigt werden, extrem einfach. Wenn wir davon ausgehen, dass alle Variablen aus einem einzelnen Zeichen bestehen und nur in PHP zulässige mathematische Operatoren verwendet werden (+, –, *, usw.), lässt sich die Aufgabe in einer einzigen Zeile realisieren:

```
$php_code = ereg_replace("([a-zA-Z])", "$\\1", $input_function);
```

Diese Zeile wandelt *m * x + b* in *$m * $x + $b* um. Wenn wir diesen regulären Ausdruck in ein wenig Programmcode einbetten und ein paar vereinfachende Annahmen machen, können wir sehr schnell einen dynamischen Funktionsplotter erstellen, wie Listing 2.6 zeigt.

```
//
// define global constants
//
define("PLOT_MIN", 0.1);
define("PLOT_MAX", 100);
define("PLOT_STEP", 0.5);
define("DIAGRAM_HEIGHT", 300);
define("DIAGRAM_HORIZON", 150);

function parse_function($in_string)
{

    // define a custom function header
    $header = "";
    $header = $header."function calculate(\$req_code, \$x)\n";
    $header = $header."{\n";
    $header = $header."eval(\$req_code);\n";

    // define a custom function footer
    $footer = "\n}\n";

    // convert all characters to PHP variables
    $out_string = ereg_replace("([a-zA-Z])", "$\\1", $in_string);

    // prepend header, create equation, and append footer
    $out_string = $header."return(".$out_string.");\n".$footer;
```

```
    // return result
    return($out_string);

}

function create_image()
{
    // export this variable
    global $color_plot;

    // we calculate the X scale based on the plot parameters
    // the diagram height is fixed as we do not check for the
    // function's extreme points
    $width = PLOT_MAX / PLOT_STEP;
    $height = DIAGRAM_HEIGHT;

    $image = imagecreate($width, $height);

    // allocate colors
    $color_backgr = imagecolorallocate($image, 255, 255, 255);
    $color_grid = imagecolorallocate($image, 0, 0, 0);
    $color_plot = imagecolorallocate($image, 255, 0, 0);

    // clear image
    imagefilledrectangle($image, 0, 0, $width - 1, $height - 1, $color_
    ➥backgr);

    // draw axes
    imageline($image, 0, 0, 0, $height - 1, $color_grid);
    imageline($image, 0, DIAGRAM_HORIZON, $width - 1, DIAGRAM_HORIZON,
    ➥$color_grid);

    // print some text
    imagestring($image, 3, 10, DIAGRAM_HORIZON + 10, PLOT_MIN, $color_
    ➥grid);
    imagestring($image, 3, $width - 30, DIAGRAM_HORIZON + 10, PLOT_MAX,
    ➥$color_grid);

    // return image
    return($image);

}

function plot($image, $x, $y)
{
    // import the color handle
```

```
    global $color_plot;
    // set these as static to "remember" the last coordinates
    static $old_x = PLOT_MIN;
    static $old_y = 0;

    // only plot from the second time on
    if($old_x != PLOT_MIN)
        imageline($image, $old_x / PLOT_STEP, DIAGRAM_HEIGHT -
        ➥($old_y + DIAGRAM_HORIZON), $x / PLOT_STEP, DIAGRAM_HEIGHT -
        ➥($y + DIAGRAM_HORIZON), $color_plot);

    $old_x = $x;
    $old_y = $y;

}

// see if we've been invoked with a function string set
if(!isset($function_string))
{
    // no, there's no function string present,
    // generate an input form
    print("<html><body>");
    print("<form action=\"".basename($PHP_SELF)."\" method=\"post\">");
    print("Function definition: <input type=\"text\"_
          ➥name=\"function_string\" value=\"(m*x+b)/(x/3)\"><br>");
    print("Required PHP code: <input type=\"text\" name=\"req_code\"
          ➥value=\"\$m = 10; \$b = 20;\"><br>");
    print("<input type=\"submit\" value=\"Parse\">");
    print("</form>");
    print("</body></html>");
}
else
{
    // translate input function to PHP code
    $parsed_function = parse_function($function_string);

    // *** NOTE: security holes! (see book contents) ***
    eval($parsed_function);

    // create image
    $image = create_image();

    // plot the function
    for($x = PLOT_MIN; $x < PLOT_MAX; $x += PLOT_STEP)
    {
        $y = calculate($req_code, $x);
```

```
        plot($image, $x, $y);
    }

    // set content type
//  header("Content-type: image/gif");
    header("Content-type: image/png");

    // send image
//  imagegif($image);
    imagepng($image);

}
```

Listing 2.6: Dynamische Funktionsparser und -plotter

Das Skript ist ausführbar. Sie können es direkt in Ihrem Browser verwenden. Beim ersten Aufruf wird es feststellen, dass Sie noch keine Funktion zum Plotten bereitgestellt haben, und ein kleines Eingabeformular anzeigen, wie Abbildung 2.13 zeigt.

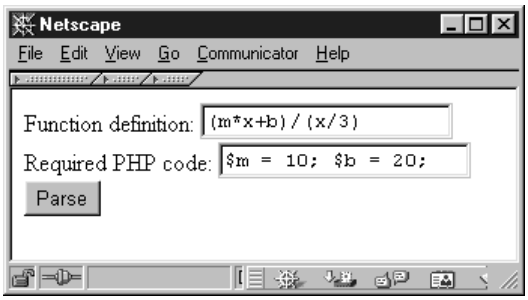

Abbildung 2.13: Eingabeformular des dynamischen Funktionplotters

Warnung: Die hier verwendete Technik, mit welcher der PHP-Code direkt mit `eval()` ausgeführt werden kann, sollte niemals (wir wiederholen: niemals!) in dieser Form in Produktionsskripten verwendet werden. Die Ausführung von Benutzerprogrammen reißt ein riesiges Sicherheitsloch in Ihre Programme. Damit könnte jeder etwas wie `system("rm -r /*")` senden, womit alle Daten gelöscht würden, auf die Ihr Webserver Zugriff hat. Wir haben es hier gemacht, weil wir uns auf die dynamische Erstellung und Ausführung von Programmcode konzentrieren wollten. Eine ausführliche Diskussion über die Möglichkeit, Ihre Skripte zu sichern (und die Ausführung von böswilligem Programmcode zu vermeiden), finden Sie in Kapitel 4 »Webanwendungskonzepte« und Kapitel 5 »Grundlegende Webanwendungsstrategien«.

Das erste Feld gibt die Funktion an, die grafisch dargestellt werden soll. Dieses Beispiel macht die Annahme, dass x die einzige Variable ist, von der die Funktion abhängt. Im zweiten Feld können Sie PHP-Code eingeben, der vor der Auswertung der Funktionsanweisung ausgeführt wird, damit Zuordnungen zu Konstanten gemacht werden können (in unserem Fall *m* und *b*).

Zum jetzigen Zeitpunkt klicken Sie einfach auf »Parse«. Abbildung 2.14 zeigt was als Nächstes erscheint.

Wie kam das Skript jetzt von dem Eingabeformular zu dieser grafischen Ausgabe? Sehen wir uns die inneren Vorgänge Schritt für Schritt an.

Nachdem Sie das Eingabeformular abgeschickt haben, beginnt das Skript mit der Ausführung der `else()`-Klausel der wichtigsten `if()`-Anweisung. Zunächst wird folgende Funktion aufgerufen:

```
// translate input function to PHP code
$parsed_function = parse_function($function_string);
```

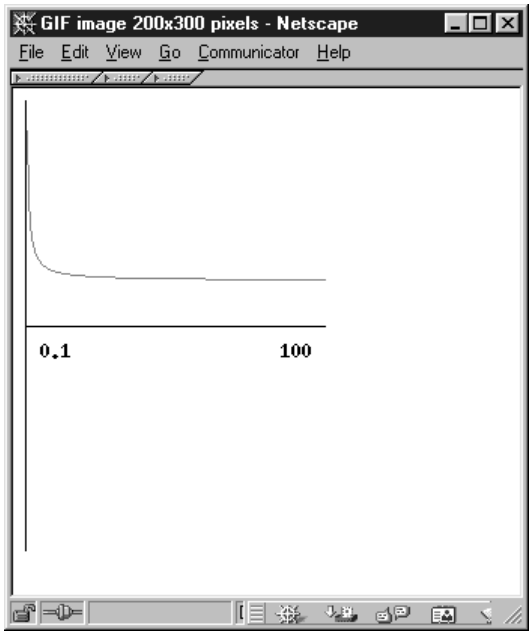

Abbildung 2.14: Beispielausgabe des Funktionplotters

`parse_function()` erzeugt den PHP-Code aus den vom Benutzer gelieferten Daten, indem es auf diese einen regulären Ausdruck anwendet. Um die mathematische Funktion einfacher nutzen zu können, wurde sie in eine kleine Funktion eingebettet, welche die entsprechenden Werte den Konstan-

ten zuordnet (durch erneute Referenz auf die Benutzereingaben). Anschließend wird die mathematische Anweisung ausgeführt und das Ergebnis an den Anrufer zurückgesendet.

Die von `parse_function()` generierte Funktion sieht für unser Beispiel *(m * x + b) / (x / 3)* folgendermaßen aus:

```
function calculate($req_code, $x)
{
    eval($req_code);
    return(($m * $x + $b) / ($x / 3));
}
```

`$req_code` enthält die Eingabe aus dem zweiten Formularfeld, in diesem Beispiel *$m = 10; $b = 20;*. Wenn wir dies mit `eval()` ausführen, erhalten wir die korrekte Variablenzuordnung für die nächste Zeile, welche bereits die gesamte Berechnung vornimmt. Und das war es!

Hinweis: Wichtige Informationen über die Anweisung `eval()` finden Sie in der vorangegangenen Warnung!

Jetzt wird die Funktion nur noch grafisch dargestellt. Die `for()`-Schleife wird für einen vordefinierten Bereich durchlaufen, und mittels `calculate()` werden bei jedem Durchgang die Y-Werte für die Kurve ermittelt.

2.7.2 Selbstmodifizierender Zähler

Ein einfaches Beispiel für selbstmodifizierende Zähler sind Trefferzähler. Normalerweise würden Trefferzähler von Protokolldateien berechnet oder aus der Datenbank ausgelesen. Eine einfachere Methode ist jedoch die Verwendung von »eingebauten« Zählern. Eingebaut bedeutet, dass sich der Programmcode und die Daten für den Zähler in derselben Datei befinden:

```
$counter = 0;
////////////////////////////////
// Do not modify above this point
////////////////////////////////

// increase counter
$counter++;

// write counter back to ourselves
$file = fopen(basename($PHP_SELF), "r+");
fputs($file, "<?\n\$counter = $counter;");

// print counter (or do something else with it)
print("$counter hits so far");
```

In der ersten Zeile wird der Zähler auf Null zurückgesetzt. Die nächste Zeile setzt ihn hoch, und nun kommt der interessante Teil: Der Programmcode öffnet seine eigene Datei und ersetzt die erste Zeile. Dies führt dazu, dass die Datei anders interpretiert wird, wenn sie von PHP das nächste Mal verarbeitet wird. Die Quelle sähe folgendermaßen aus:

```
$counter = 1;
////////////////////////////////
// Do not modify above this point
////////////////////////////////

// increase counter
$counter++;

// write counter back to ourselves
$file = fopen(basename($PHP_SELF), "r+");
fputs($file, "<?\n\$counter = $counter;");

// print counter (or do something else with it)
print("$counter hits so far");
```

Jetzt setzt die erste Zeile `$counter` auf 1 und nicht auf 0. Bei jedem weiteren Durchlauf durch die Datei wird sich die erste Zeile ändern und die Anzahl der Treffer angeben, die es für die Datei bisher gab.

Hinweis: Gleichzeitige Zugriffe mehrerer Anrufer kann dieses Programm nur schwer bewältigen. Es könnte vorkommen, dass zwei PHP-Prozesse die Datei zur selben Zeit lesen (und gleichzeitig in sie schreiben), was zu einem falschen Trefferzähler führen würde. In einer Umgebung mit umfangreichem Datenverkehr sollten Sie diese Technik nur mit entsprechenden Verarbeitungssperren einsetzen.

2.8 Zusammenfassung

In diesem Kapitel haben Sie eine Menge über die erweiterte Syntax von PHP und gute Programmierpraktiken erfahren. Wir haben Ihnen gezeigt, wie Sie Konstanten mit `define()` erzeugen. Anschließend sind wir auf die kniffligeren Aspekte von Arrays eingegangen und haben Ihnen erklärt, dass Sie `list()/each()` zur Abarbeitung von Hash-Codes verwenden sollten. Wir erläuterten die Funktionen der objektorientierten Programmierung in PHP, zeigten, wie und wann Sie diese verwenden, und sagten Ihnen, wann Sie besser bei der prozeduralen Programmierung bleiben sollten. Da PHP eine interpretierte Sprache ist, lässt sie viele Funktionen zu, die sich in konventionell

kompilierten Programmiersprachen nur schwer implementieren ließen: variable Variablen und Funktionen, selbstmodifizierender Code und Laufzeitauswertung von Quellcode. Mit diesem Wissen sind Sie gut gerüstet für die erweiterte PHP-Programmierung und einen großen Schritt weiter auf Ihrem Weg zum PHP-Experten.